Algorithms for Domination Problems on Temporal Graphs

Mees Verheije 4144805

Supervisor: Prof. dr. Hans Bodlaender

Second Supervisor: Dr. Johan van Rooij

Abstract

Temporal graphs are graphs whose edge set can change on a discrete set of τ ordered time steps. We research temporal domination problems. We consider several static domination problems which describe *how* a vertex can be dominated, and several temporal domination requirements which describe *when* and *how often* a vertex should be dominated. By combining static domination problems with temporal domination requirements, we obtain temporal domination problems.

For the Temporary Domination requirement, each vertex needs to be dominated in at least one time step, while the Permanent Domination requirement requires them to be dominated in all time steps. k-Fold Domination generalizes this: each vertex should be dominated on at least k time steps. Periodic Domination requires each vertex to be dominated at least once every θ time steps for some period θ. Evolving Domination instead asks for a separate domination function for each time step. Marching Domination requires each vertex to be dominated in every time step, but allows us to change the domination function in between time steps, simulating the movements of armies protecting points of interest (vertices).

Each of these temporal domination requirements can be combined with static domination problems. We apply these six temporal domination requirements to three static domination problems: Dominating Set, Roman Domination and Weak Roman Domination to define 18 temporal domination problems. We provide exact exponential and parameterized algorithms that solve all of these problems, except for the permanent, periodic and k-fold variants of Weak Roman Domination.

Furthermore, we also give definitions for the family of static domination problems Defense-Like Domination and their natural temporal counterpart: Temporal Defense-Like Domination.

Utrecht University

Graduate School of Natural Sciences
Utrecht University
Netherlands

Contents

1 Introduction 2

I Introduction to Temporal Graphs and Domination Problems 3

2 Preliminaries 4
- 2.1 Temporal Graphs 4
- 2.2 Layers 5
- 2.3 Neighbourhoods 5
- 2.4 Tree Decompositions 6

3 Domination Problems 7
- 3.1 Dominating Set 7
- 3.2 Roman Domination 9
- 3.3 Weak Roman Domination 9
- 3.4 Defense-Like Domination 10

4 Temporal Domination Requirements 13
- 4.1 Domination Rules 14
- 4.2 Temporary Domination 14
- 4.3 Permanent Domination 15
- 4.4 Periodic Domination 16
- 4.5 Evolving Domination 16
- 4.6 k-Fold Domination 17
- 4.7 Marching Domination 18
- 4.8 Temporal Defense-Like Domination 19

II Temporal Domination Algorithms 21

5 Temporary Domination 22

6 Permanent Domination 23
- 6.1 Parameterized Algorithm Structure 23
- 6.2 Permanent Dominating Set 32
- 6.3 Permanent Roman Domination 39

7 Periodic Domination 43

8 Evolving Domination 44
- 8.1 Maintenance Algorithms 44
- 8.2 Using the Trivial Solution 47

9 k-Fold Domination 47
- 9.1 k-Fold Dominating Set 47
- 9.2 k-Fold Roman Domination 52

10 Marching Domination 54
- 10.1 Positive Instance Based Dynamic Programming 56
- 10.2 Improvements for Practical Scenarios 59

III Conclusion 65

11 Obtained Results 66

12 Future Work 67

1 Introduction

Graphs are mathematical structures that capture the nature of networks. As simple as graphs are, consisting only of a collection of vertices and edges or arcs connecting them, they are capable of simulating many different kinds of networks, such as social networks or infrastructural networks of roads and railways and many, many other types of networks. For some applications, however, the bare "skeleton" (meaning the vertices and edges or arcs) of a graph is not enough, we instead need to attach extra information to the vertices and edges or arcs. For a road network where arcs represent roads between two locations (vertices), it might be necessary to attach information to the arcs denoting the length of those roads for some applications. By adding this extra information we make the graph suitable for solving certain problems. But graphs have one noticeable shortcoming: they only represent one static situation. What if a road is demolished or a new road is built? What if a friendship ends or a new one is established? Or what if certain roads are only open during certain hours? For those situations, we can make multiple graphs, each one representing the network at a different time. Such a structure consisting of an ordered collection of graphs sharing the same vertices, is called a *temporal graph*.

Temporal graphs are graphs where edges can appear or disappear on every time step, in a given set $T = \{1, \cdots, \tau\}$ of time steps for some $\tau \in \mathbb{N}$. The term *static graph* will be used to refer to traditional graphs that do not change. Temporal graphs can be used in practical applications to chart spreading processes (for example disease or information) with changing connectivity. Despite their applicability, temporal graphs are still a very little studied subject with much left to be discovered. Many static graph problems that are very well studied, lack even a single exact algorithm for their temporal variants. Temporal graphs will be defined properly in Section 2.1.

Temporal graphs can help to solve problems that, on static graphs, would be impossible. Most graph theory problems on static graphs can be translated to problems on temporal graphs. In fact, one static graph problem may have multiple temporal interpretations. A path, for example, can mean many different things in a temporal graph. Generally we are talking about a path where each step over an edge is taken in a different time step, thus stopping us from using certain edges at certain times. It may also mean that we can take multiple steps at once, allowing us to stay in one time step as long as needed before proceeding to the next time step. Going back in time is, of course, not possible. Whatever definition we choose for a path, a shortest path can also mean many different things. Do we mean a path that gets from the start vertex to the end vertex in the least amount of edges, or in the least amount of time steps? In the latter case: can we start at any time step t (whichever t allows us to make the path with the least amount of time steps starting from t), or do we have to start from time step 1? Simply put: for (almost) every static graph problem, there are multiple temporal graph problems. In most cases, these temporal problems have not been explored much in literature, even when their static counterparts are some of the most well known problems in graph theory.

In this computing science master's thesis, we are going to explore domination problems in temporal graphs. In particular, we are going to focus on DOMINATING SET, ROMAN DOMINATION, a problem closely related to DOMINATING SET and WEAK ROMAN DOMINATION. Like the shortest path problem, these problems do not have an obvious single catchall temporal interpretation. The static variant of DOMINATING SET asks, given a graph $G = (v, E)$, for a vertex set $D \subset V$ that "dominates" each vertex. This means that each vertex $v \in V$ of the graph has to be in that vertex set: $v \in D$, or must have a neighbour u in it: $u \in D$. How can we translate this to a temporal graph problem? What does it mean to "dominate" vertices in a temporal graph. One interpretation is this: Find a vertex set that dominates every vertex *on every time step*. Another possibility is to ask for a vertex set that dominates every vertex *on at least one time step*.

The thesis consists of three parts. The first part will focus on introducing temporal graphs, the static domination problems, and the ways in which we can translate these problems to the temporal setting. The definitions needed for temporal graphs are presented in Section 2. The static problems DOMINATING SET, ROMAN DOMINATION and WEAK ROMAN DOMINATION will be explained in detail in Section 3, along with another static domination problem called DEFENSE-LIKE DOMINATION. Section 4 will present several ways to translate static domination problems to the temporal case. The second part of the thesis will focus on designing algorithms that solve some of these temporal domination problems. The final part concludes the thesis by reflecting on obtained results in Section 11 and provides suggestions for future work in Section 12.

Part I

Introduction to Temporal Graphs and Domination Problems

The first part of the thesis is meant to provide the knowledge which is required for the second part. Specifically: the principles of temporal graphs, the domination problems Dominating Set, Roman Domination, Weak Roman Domination and Defense-Like Domination and finally how one can turn these static problems into temporal problems. The goal of this part is to introduce several temporal domination problems, while the second part will focus on finding solutions for them.

2 Preliminaries

Let $\mathbb{N}$ and $\mathbb{N}_0$ be the natural numbers without and with 0 respectively. We denote subsets with $\subset$ and do not use strict subsets. Whenever we are talking about a graph $G = (V, E)$ we generally use m to denote $|E|$ and n to denote $|V|$.

2.1 Temporal Graphs

A *temporal graph* is, in essence, a graph that can change. For a given vertex set, edge set and a set of time steps, a temporal graph defines when each edge exists and when it does not. To avoid confusion between temporal graphs and non-temporal graphs, we will use the word *static* whenever we wish to specify that we are in fact talking about a non-temporal graph. We will also use the terms *temporal problem* and *static problem* when referring to a problem on a temporal graph, or a static graph respectively.

Definition 2.1. *A* ***temporal graph*** $\mathcal{G} = (V, E, \tau, \lambda)$ *is the combination of a vertex set* V*, an edge set* E *on those vertices, a lifetime* $\tau \in \mathbb{N}$ *and a function* $\lambda : E \to \mathcal{P}(T_\tau)/\{\emptyset\}$ *where* $T_\tau = \{1, \cdots, \tau\}$ *is called the* ***timeline****.*

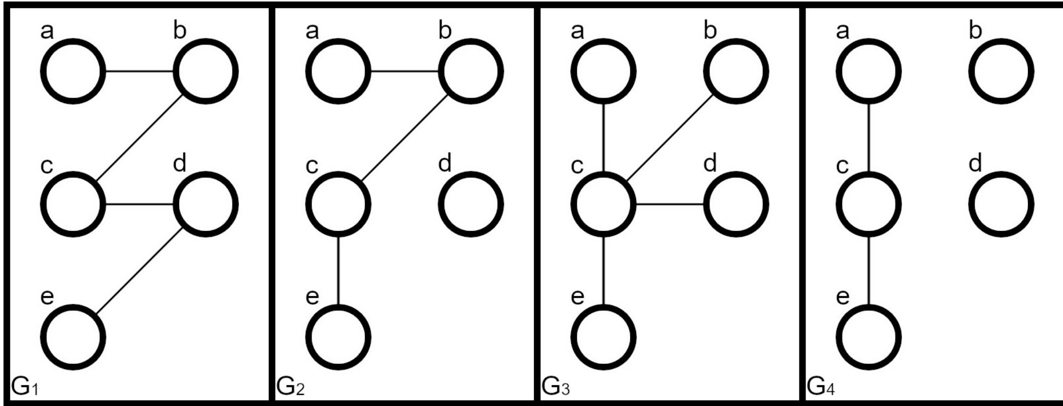

Figure 2.1: A temporal graph $\mathcal{G}$ with lifetime $\tau = 4$. Each picture shows a different layer of $\mathcal{G}$ as explained in Definition 2.3.

We write T instead of T_τ whenever it is clear which τ we are referring to. A temporal graph induces a graph that exists on the time steps given by T. The vertices of this graph are the same for each time step. However, on each time step t, the set of edges can be different. The function λ maps each edge e to a set $\lambda(e) \subset T$ of time steps on which e "exists", or is "available". On all other time steps, e does not exist, or is "unavailable". The empty set is excluded from λ's codomain, since $\lambda(e) = \emptyset$ would imply that an edge never exists, in which case we need not have included it in E in the first place. At the base of each temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ is a static graph (V, E) which defines the underlying structure. This graph is called the *underlying graph.*

Definition 2.2. *Let* $\mathcal{G} = (V, E, \tau, \lambda)$ *be a temporal graph. The* ***underlying graph*** *of* $\mathcal{G}$ *is the static graph* $\mathcal{G}_\downarrow = (V, E)$*.*

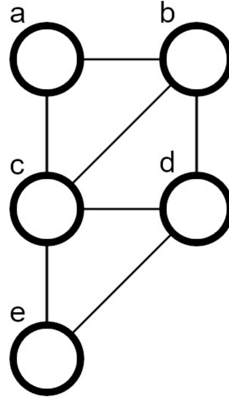

Figure 2.2: The underlying graph corresponding to the temporal graph in Figure 2.1.

2.2 Layers

It can be useful to refer to a specific instance in time. A temporal graph is described as a *changing* graph. To visualize these changes, we use *layers*.

Definition 2.3. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and $t \in T$ a time step. $E_t := \{e \in E : t \in \lambda(e)\}$ is the **timed edge set** of edges available at time step $t \in T$. Then $\mathcal{G}_t := (V, E_t)$ is the **layer** of $\mathcal{G}$ at time step t.*

The layers allow us to visualize what $\mathcal{G}$ looks like on a specific time step. A layer $\mathcal{G}_t$ is simply $\mathcal{G}_\downarrow$ limited to the edges available on time step t. The vertices and edges in $\mathcal{G}_t$ are not unique to that layer. They can also appear in other layers. Sometimes, however, it is useful to regard each layer as a completely separate graph instead of a subgraph of $\mathcal{G}_\downarrow$. We can make time stamped copies of the vertices and edges called *vertex appearances* and *edge appearances*.

Definition 2.4. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and $t \in T$ a time step. Let $v \in V$ be some vertex and $e \in E$ be some edge such that $t \in \lambda(e)$, then time stamped vertex v^t is the **vertex appearance** of v in time step t and time stamped edge e^t is the **edge appearance** of e in time step t.*

Using the notion of vertex and edge appearances, we can define, for each time step t, the vertex set V^t of vertex appearances and the edge set E^t of edge appearances of edges that exist in that moment. We can also use these sets to define an alternative type of layer that is independent of V and E.

Definition 2.5. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and $t \in T$ a time step. $V^t := \{v^t : v \in V\}$ is the **vertex appearance set** of time stamped vertices in time step t and $E^t := \{e^t : e \in E, t \in \lambda(e)\}$ is the **edge appearance set** of time stamped edges available at time step $t \in T$.*

We will sometimes denote the layers $\mathcal{G}_t$ as G_t and the underlying graph $\mathcal{G}_\downarrow$ as $G_\downarrow$. The reason for using regular letters instead of calligraphy here, is to emphasise that these graphs are static graphs, not temporal graphs. Using the concept of layers, we can also denote a temporal graph as $\mathcal{G} = (G_1, \cdots, G_\tau)$.

2.3 Neighbourhoods

Let $G = (V, E)$ be a static graph, then the *open neighbourhood* $N_G(v) = \{u : \{v, u\} \in E\}$ of v is the set of neighbours of v in G. Let $N_G[v] = N_G[v] \cup \{v\}$ be the *closed neighbourhood* of v that includes v and all its neighbours in G. We omit the subscript when it is clear what static graph we refer to. When it is clear what temporal graph we are referring to, we will denote the open and closed neighbourhoods of vertex v in layer G_t as $N_t(v)$ and $N_t[v]$ respectively. We will denote the open and closed neighbourhoods of vertex v in the underlying graph $G_\downarrow$ as $N(v)$ and $N[v]$ without any subscripts, unless it is unclear what graph we are referring to, in which case we will denote them as $N_{G_\downarrow}(v)$ and $N_{G_\downarrow}[v]$. The open and closed neighbourhoods for a vertex appearance v^t in $G^t := (V^t, E^t)$ are denoted as $N(v^t)$ and $N[v^t]$ respectively. The subscripts are left out since the time stamp should make it clear what graph we are referring to.

2.4 Tree Decompositions

A useful tool for dynamic programming on graphs that we will be using, is the concept of *tree decompositions*. This is a tree associated with a static graph, that helps us understand the structure of a graph better. In particular it shows how vertices in the graph are connected to each other.

Definition 2.6. *Let $G = (V, E)$ be a graph. A* ***tree decomposition*** *$\mathcal{T} = (I, T)$ for G is a tree where each node $i \in I$ is associated with a bag $X_i \subset V$ such that the following conditions are satisfied:*

- $\bigcup_{i \in I} X_i = V$.
- *For every edge $\{v, u\} \in E$ there is an index $i \in I$ such that $\{v, u\} \subset X_i$.*
- *Let $v \in V$, $i, j \in I$ such that $v \in X_i$ and $v \in X_j$. Then for every index k on a path from i to j on the tree T we have that $v \in X_k$.*

The ***width*** *of $\mathcal{T}$ is $\max_{i \in I} |X_i| - 1$. The minimum width over all possible tree decompositions is called the* ***treewidth*** *of G, denoted $tw(G)$.*

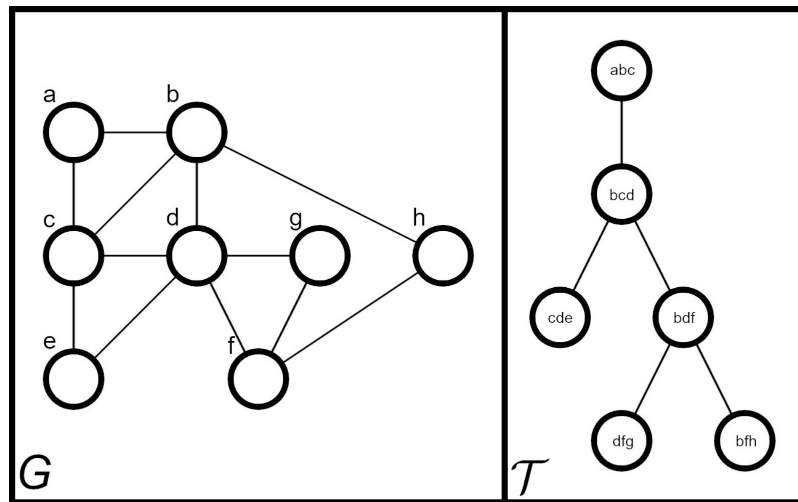

Figure 2.3: The left picture shows a static graph G. The right picture shows a tree decomposition $\mathcal{T}$ for G of width 2.

Tree decompositions, and even minimum tree decompositions, are not unique. There can be many different for one graph. To make it easier to perform dynamic programming on tree decompositions, we can limit our algorithms to a specific type of tree decomposition with some desirable properties. These nicer tree decompositions are, fittingly, called *nice tree decompositions* [24].

Definition 2.7. *A* ***nice tree decomposition*** *of G is a tree decomposition $\mathcal{T}$ such that T is a rooted tree with only the following types of nodes:*

- *Leaf nodes i have one vertex $|X_i| = 1$.*
- *Introduce nodes i have one child node j, and $X_i = X_j \cup \{v\}$ for some vertex $v \in V$.*
- *Forget nodes i have one child j, and $X_i = X_j / \{v\}$ for some vertex $v \in V$.*
- *Join nodes i have two children j and k, and $X_i = X_j = X_k$.*

Furthermore, the root node r is always a forget node or join node with bag $X_r = \emptyset$.

Nice tree decompositions are easier to use because the difference between two neighbouring bags are very small. This makes them easier to work with. For example it can be very useful to use dynamic programming algorithms on a nice tree decomposition. We use such techniques in Sections 6 and 9.

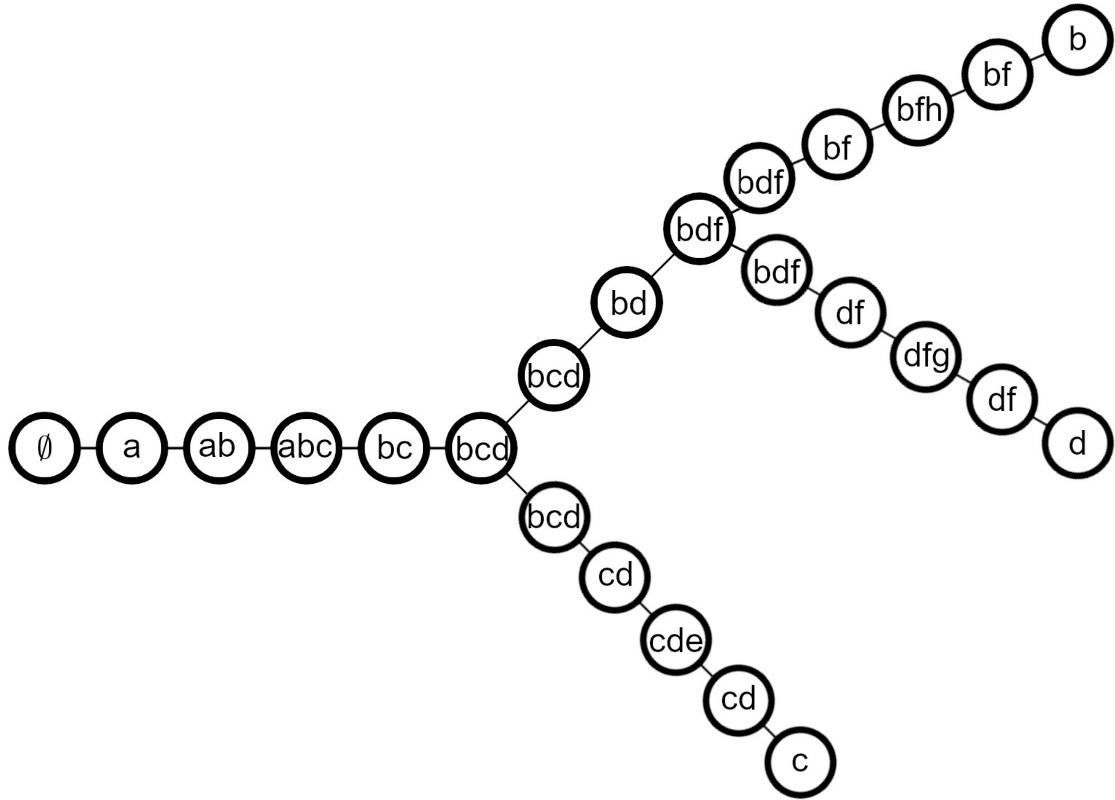

Figure 2.4: A nice tree decomposition of the graph from Figure 2.3 with the same width. A natural way to expand the tree decomposition from that figure to a nice tree decomposition.

3 Domination Problems

Domination in graphs refers to some structure (a vertex set, or a labelling function in most cases) that "covers" or "dominates" all the vertices (or in some cases all edges) in some way. For example: we may choose certain vertices that dominate their neighbours, or the edges connected to them. Domination problems ask for a a specific type of domination structure that dominates all vertices or all edges, and it has to be of minimum size. In this section we will introduce static problems first, and then discuss how they can be adapted to the temporal setting.

3.1 Dominating Set

Perhaps the most well-known domination problem is Dominating Set. This problem asks us to choose a subset of vertices that cover themselves and their neighbours. We want to choose this subset in such a way that it covers all vertices of the graph, and we want it to be of minimum size.

Definition 3.1. *Let $G = (V, E)$ be a static graph. A* ***dominating set*** *on G is a subset $D \subset V$ of vertices such that each vertex $v \in V$ is either in D or has a neighbour in D.*

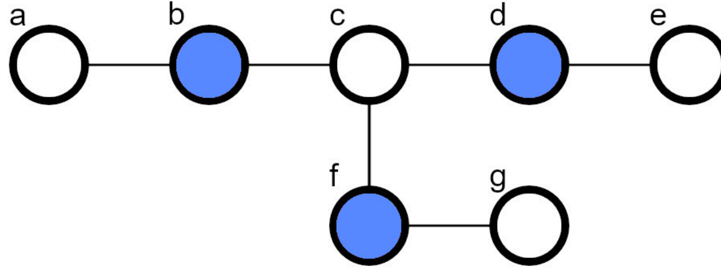

Figure 3.1: The blue vertices form a dominating set of size 3 on a static graph G with $n = 7$.

Using the definition for a dominating set, we can now define the first problem.

Problem 3.1. Dominating Set*: For a given static graph $G = (V, E)$, find the size of a minimum dominating set.*

Two other static domination problems that we discuss in this thesis, are the Roman Domination problem and the Weak Roman Domination problem. These problems are similar to Dominating Set, but they make use of a *domination function* instead of a dominating vertex set. A domination function is a labelling function that assigns a label (a non-negative integer) to each vertex.

Definition 3.2. *Let G be a graph (it can be static or temporal) with vertex set V. A **domination function** on G is a function $f : V \to \mathbb{N}_0$. The **size** of f is the sum of the labels $|f| = \sum_{v \in V} f(v)$.*

To create unity in the way we refer to domination problems, we propose an alternative definition for the Dominating Set problem that is more in line with the definitions for the Roman Domination problem and the Weak Roman Domination problem. This alternative definition will make use of a domination function instead of a dominating vertex set. We first define a domination function that is the equivalent of the dominating set.

Definition 3.3. *Let $G = (V, E)$ be a static graph. A **standard domination function** (sdf) is a domination function $f : V \to \{0, 1\}$ such that for each vertex $v \in V$ there is a vertex $u \in N[v]$ in the closed neighbourhood of v such that $f(u) = 1$. In this case we say that u **dominates** v.*

An sdf and a dominating set are essentially two different ways of the same construction. For some vertex $v \in V$, being included in a dominating set D is the same as being given label $f(v) = 1$ in some sdf f. For any sdf f, we can define the corresponding dominating set $D_f = \{v \in V : f(v) = 1\}$. For some dominating set D, we can also define the corresponding sdf

$$f_D(v) = \begin{cases} 1 & \text{if } v \in D \\ 0 & \text{otherwise} \end{cases} .$$

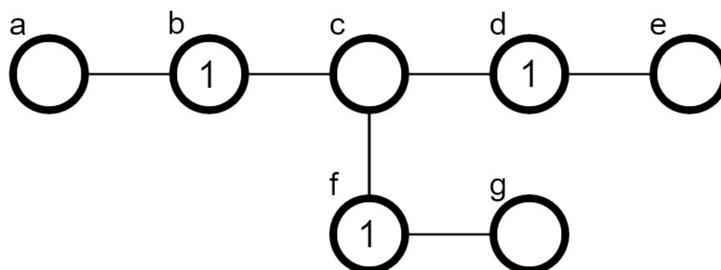

Figure 3.2: The same dominating set as in Figure 3.1 but presented as an sdf.

Using the notion of an sdf, we can give an alternative definition for Dominating Set.

Problem 3.2. Dominating Set*: For a given static graph $G = (V, E)$, find the size of a minimum sdf.*

3.2 Roman Domination

In the article "Defend the Roman Empire!" [29], Stewart describes an old military strategy from the Roman empire in the time of emperor Constantine: Not every city needs to have its own army. Instead, we can have cities without armies as long as they have at least one neighbouring city with two armies and thus an army to spare. He made a mathematical model based on this strategy. In this problem, vertices represent cities and edges represent roads between the cities. The concept was formalized further by Cockayne et al. [11], who studied the graph theoretic properties of the *Roman domination function*. The labels of a Roman domination function represent the number of armies present in a city.

Definition 3.4. *Let $G = (V, E)$ be a static graph. A* ***Roman domination function*** *(rdf) on G is a domination function $f : V \to \{0, 1, 2\}$ such that for each vertex $v \in V$ we have: $f(v) > 0$ or v has a neighbour $u \in N(v)$ with $f(u) = 2$. In this case we say u* ***dominates*** *v.*

An rdf gives a specific role to each vertex v. Some dominate only themselves ($f(v) = 1$), some dominate themselves and all their neighbours ($f(v) = 2$) and some have to be dominated by other vertices ($f(v) = 0$).

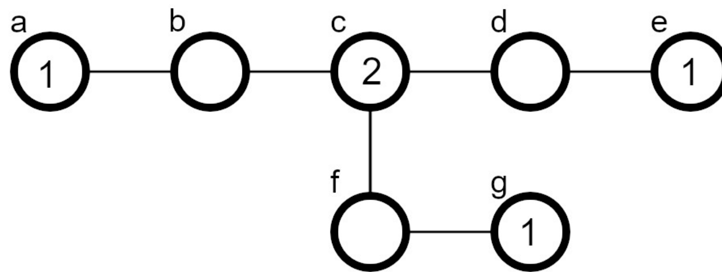

Figure 3.3: An rdf on the graph from Figure 3.1 of size 5. If we simply took the sdf from Figure 3.2 and set the labels to 2 instead of 1, we would get an rdf of size 6. By smartly using the properties of a Roman domination, we can sometimes obtain results that are better than twice the size of a minimum sdf.

Using the notion of an rdf, we can now define the next problem:

Problem 3.3. Roman Domination*: For a given static graph $G = (V, E)$, find the size of a minimum rdf.*

3.3 Weak Roman Domination

Elaborating further on the concept of Roman Domination, Henning [20] introduced the problem Weak Roman Domination, a version of Roman Domination with fewer restrictions. We use terminology from [10].

Definition 3.5. *Let $G = (V, E)$ be a static graph and let $f : V \to \mathbb{N}_0$ be a domination function. A vertex v is* ***secured*** *by f if $f(v) > 0$ and is* ***defended*** *by f if there is a vertex $u \in N[v]$ with $f(u) > 0$, in which case we say that u* ***defends*** *v, and* ***undefended*** *otherwise.*

We can take the idea behind Roman domination even further by considering the following situation: City A, that does not have its own army, can be defended even by a neighbouring city B that has only one army, as long as moving that army from B to A would not leave other cities undefended. That is why a vertex is considered defended if it has a neighbour u with $f(u) = 1$. But as stated, moving an army from one city to another may leave other cities undefended as per Definition 3.5. To formalize this, we need the following notion. Let $f_{u \to v}$ be the domination function that we obtain by taking f and *moving an army from u to v*.

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

Note that u and v can be the same vertex, in which case $f = f_{u \to v}$.

Definition 3.6. *Let $G = (V, E)$ be a static graph and $f : V \to \{0, 1, 2\}$ a domination function. A vertex $v \in V$ is* ***safely defended*** *by f if there is a vertex $u \in N[v]$ with $f(u) \geq 1$ and all vertices that are defended by f are also defended by $f_{u \to v}$. In this case we also say that u* ***safely defends*** *v. A domination function f that safely defends all vertices $v \in V$ is called a* ***weak Roman domination function*** *(wdf).*

When vertex u safely defends vertex v, we also say that u *dominates* v.

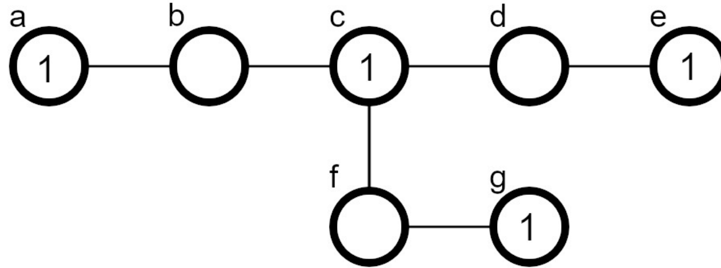

Figure 3.4: A wdf on the graph from Figure 3.1 of size 5. The sdf from Figure 3.2 defends all vertices, but does not *safely* defend a, e and g, so it is not a wdf. However, taking the rdf from Figure 3.3 and changing the label of c from 2 to 1 allows us to safely defend all vertices while still decreasing the size compared to taking a minimum sdf and changing the labels to 2.

Definition 3.7. *Let $G = (V, E)$ be a static graph, $v \in V$ a vertex and $f : V \to \{0, 1, 2\}$ a domination function. If f defends v but does not safely defend v, we say f* ***unsafely defends*** *v. In this case $f(v) = 0$ and for every secured neighbour u of v we have $f(u) = 1$ and $f_{u \to v}$ leaves some neighbour $w \in N(u)$ undefended. We say w is* ***weakly defended by*** *u and* ***due to*** *v.*

The definition for a wdf induces the next problem:

Problem 3.4. Weak Roman Domination*: For a given static graph $G = (V, E)$, find the size of a minimum wdf.*

3.4 Defense-Like Domination

We can take the idea of Weak Roman Domination one step further. In his master's thesis [32], Veldhuizen gives a definition for Defense-Like Domination which is a generalized family of problems based on Weak Roman Domination. We will use much of the same terminology from [32] and from [10].

We have this notion of a vertex u "defending" another vertex v because we *could* move an army from u to v in case of an attack. These attacks however, are completely theoretical. We could instead imagine a problem where the attacks are real. This does not lead to a single new problem but rather to a family of problems that are called Defense-Like Domination problems. Defense-Like Domination (DefDom) problems are best described as a sort of board game where the game board is represented by a static graph $G = (V, E)$. Before we define the problem, we first need to introduce a few parameters:

k: The number of turns.

a: The number of attacked vertices per turn.

h: The number of armies that can move in one turn.

c: The "capacity" of the vertices: the maximum number of armies per vertex.

s: The maximum number of steps an army can take.

These parameters are part of what defines the problem at hand, and are thus chosen before the "game" begins. In the game, a number of turns are played on the game board which is a static graph

$G = (V, E)$. This game has two players, the *defender* and the *attacker*. The defender gets to pick a domination function that describes where the defending armies are stationed at the start of the game and the attacker gets to pick vertices that will be attacked on every turn. Each turn starts with the attacker revealing what vertices they are going to attack. Then, the defender gets to make their move. The defender can move their armies, and has to do so in such a way that each vertex that is under attack contains at least one army. The movement of these armies has to conform to the rules imposed by the parameters.

To formalize the movements of armies, we need to understand what a single movement looks like, and how we can chain such movements together. In that regard, we introduce the notion of an *intermediate domination function* or simply an *intermediate function*. Such a function $g : V \to \mathbb{N}_0^2$ maps any vertex to a vector with two coordinates which reflect what the distribution of armies looks like after a particular move has been made. For some vertex $v \in V$, the first coordinate, $g(v).x$ represents the number of armies present on v, like a the value $f(v)$ for some domination function $f : V \to \mathbb{N}_0$. The second coordinate $g(v).y$ of $g(v)$ represents the number of *incoming* armies: the number of armies that will arrive on the vertex by the end of the defenders move.

Definition 3.8. *Let $G = (V, E)$ be a static graph, $g : V \to \mathbb{N}_0^2$ an intermediate function on G and k, a, h, c and s the parameters. If the minimum number of edges on a path from u to v (the distance between u and v) in G is $d_{u,v} \leq s$ and $f(u).x > 0$, then the intermediate function $g_{u \to v}$ defined as*

$$g_{u\to v}(w) = \begin{cases} (g(w).x \quad , g(w).y + 1 \quad) & \text{if } w = v \neq u \\ (g(w).x - 1 \quad , g(w).y \quad) & \text{if } w = u \neq v \\ (g(w).x \quad , g(w).y \quad) & \text{otherwise} \end{cases}$$

is called a ***partial move*** *for g.*

A partial move represents the defensive player moving a single army from one vertex u to another vertex v, crossing a maximum of s edges. The army does not immediately arrive on v: the x coordinate remains the same for now. Instead, the number of incoming armies for v increases. The reason for using this construction instead of simply calling the domination function $f_{u \to v}$ (as defined in Section 3.3) a partial move, is that the latter option omits the information of which armies have already been moved this turn. This would allow the defender to move an army multiple times in the same turn. To make a valid move, we perform a number of partial moves consecutively before finalizing it by making all incoming armies arrive, all while obeying the capacity restriction.

Definition 3.9. *Let $G = (V, E)$ be a static graph, $f : V \to \mathbb{N}_0$ a domination function on G and k, a, h, c and s the parameters. Let $g_0, g_1, \cdots, g_h$ be a sequence of intermediate functions such that $g_0(v) = (f(v), 0)$ for each vertex $v \in V$ and g_i is a partial move for g_{i-1} for $1 \leq i \leq h$. Let $f' : V \to \mathbb{N}_0$ be the domination function such that $f'(v) = g_h(v).x + g_h(v).y$ for all vertices $v \in V$. If $f'(v) \leq c$ for all vertices $v \in V$, then f' is a* ***valid move*** *for f.*

On each turn i, the attacker reveals a set of vertices A' that is going to be attacked. This set is simply referred as an *attack*. The defender has to protect all vertices that are under attack by making a valid move for the current domination function f such that each vertex that is under attack has at least one army. Such a valid move *defends against* the attack.

Definition 3.10. *Let $G = (V, E)$ be a static graph, f a domination function, A' an attack and k, a, h, c and s the parameters. f* ***defends against*** *attack A' if $f(a) > 0$ for each vertex $a \in A'$.*

Let us explain the game once more, this time by using the mathematical terms we have just introduced. The game board is a static graph $G = (V, E)$, on which k turns are played by the two players: the defender and the attacker. At the start of the game, the defender chooses an initial domination function f such that $f(v) \leq c$ for every vertex $v \in V$. The attacker gets to pick an *attack sequence* $A = (A_1, \cdots, A_k)$, a sequence of vertex subsets $A_i \subset V$ that represent the vertices that will be attacked at turn i, with $|A_i| \leq a$ for $1 \leq i \leq k$. Each turn i starts with the attacker revealing attack A_i. Then, the defender has to make a new domination function f_i that is a valid move for the previous domination function f_{i-1} such that f_i defends against A_i. If this is not possible, the attacker wins. If, at the end of turn k, the attacker has not yet won, the defender wins. In this case, the initial domination function f *defends against* attack sequence A.

Definition 3.11. *Let $G = (V, E)$ be a static graph, $f : V \to \mathbb{N}_0$ a domination function, $A = (A_1, \cdots, A_k)$ an attack sequence and k, a, h, c and s the parameters. Initial domination function f*

defends against *attack sequence A if there exists a sequence $f_0, f_1, \cdots, f_k$ of domination functions such that $f_0 = f$ and f_i is a valid move for f_{i-1} that defends against A_i for every $1 \le i \le k$.*

The attack sequence is hidden from the defender and only revealed bit by bit. Since they cannot know what vertices are going to be attacked before the domination function is chosen, a defender needs to choose an initial domination function that can defend against any attack sequence in order to guarantee their victory.

Definition 3.12. *Let $G = (V, E)$ be a static graph and k, a, h, c and s the parameters. A domination function $f : V \to \mathbb{N}_0$ that defends against any attack sequence is called a* ***defense-like domination*** *(dld). The* ***size*** *of a dld f is the sum of the values $\sum_{v \in V} f(v)$.*

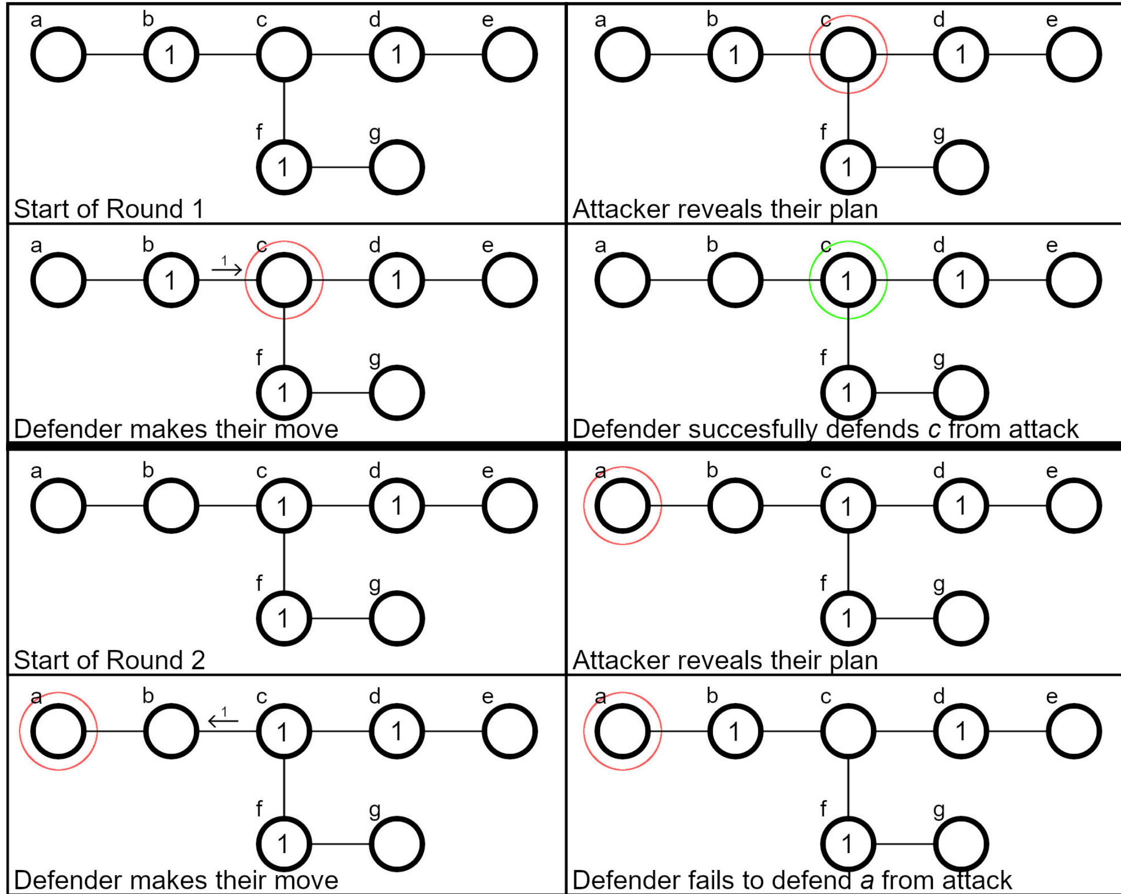

Figure 3.5: An example of a defense-like domination game being played with $k = 2$, $a = 1$, $h = 1$, $c = \infty$ and $s = 1$. The defender uses the set up as in the sdf from Figure 3.2, an attempted solution of size 3. The game is won by the attacker because the defending armies can be forced to move away, leaving another vertex undefended. It is impossible for the defender to win this game with 3 armies.

Note that for the parameters chosen in Figures 3.5 and 3.6, a dld is equivalent to a wdr. Both a dld with these parameters and a wdr have to be capable of defending against a first attack without leaving another vertex open for a second one.

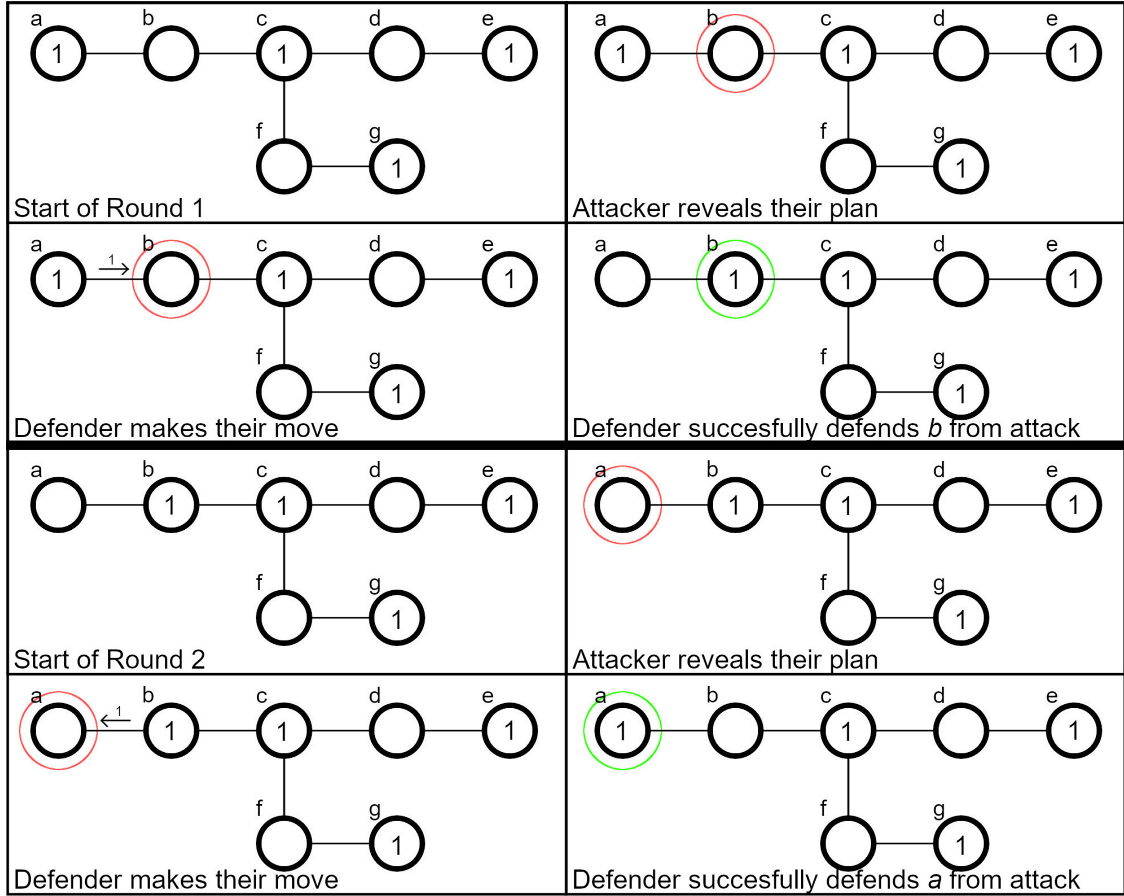

Figure 3.6: An example using the same graph and parameters as Figure 3.5, but with a different strategy by the defensive player. By adding a fourth army and placing the armies like the wdf from Figure 3.4, it becomes impossible for the attacker to win, if the defender uses the following strategy: always keep an army on c, and use three other armies to each guard two neighbouring vertices, one army for a and b, one army for d and e and one army for f and g. If a vertex is attacked, there is always an army that can defend it without leaving another vertex unguarded.

Of course, the defender could just create a dld by placing a single army on every vertex and sit back while every attack is foiled. This would not make for an interesting problem of course.

Problem 3.5. Defense-Like Domination: *Given a graph $G = (V, E)$ and parameters k, a, h, c and s, find a dld of minimum size that satisfies the constraints imposed by the parameters.*

4 Temporal Domination Requirements

Every static problem can be considered in a temporal setting. One simple way to do this, is by simply applying the static problem to a certain layer, or to the underlying graph. But the way in which a static problem can be applied to a temporal graph is not unique, and most of the time there is no obvious "best" way to do it. Consider the example of the shortest path problem given in the introduction of the thesis. On a static graph, a shortest path is simply a path with the least number of edges. The concept of a shortest path is connected to time. A shortest path after all, is a path that takes the least amount of time to travel. But in a temporal graph, where time becomes part of the problem itself, a path with the least number of edges and a path that takes the least amount of time to travel are no longer necessarily the same thing. Since edges are not always available in a temporal graph, taking a path with a minimum number of edges may require us to wait several time steps for a specific edge to become available, while a longer path where each edge is available immediately could have brought us to our destination faster.

The goal of this thesis is to introduce a number of temporal domination problems derived from static domination problems, and to find efficient exact algorithms that solve these problems. The time complexity of these algorithms is prioritized over the space complexity. In this section we will

derive a number of temporal counterparts to the static domination problems we have introduced so far. Each of the temporal graph problems (except for TEMPORAL-DEFENSE LIKE DOMINATION) is obtained by combining a static graph problem such as DOMINATING SET, ROMAN DOMINATION and WEAK ROMAN DOMINATION, which describes *how* a vertex can be dominated, with a temporal domination requirement which describes *when* and *how often* a vertex should be dominated.

To formalize this, we will first define the notion of *domination rules*: a description of how a vertex is dominated in a static graph. Then we will define six different temporal domination requirements. Finally, we will also describe a natural way of adapting DEFENSE-LIKE DOMINATION to temporal graphs.

To help visualize the different temporal domination variants, we provide an example graph on which all DOMINATING SET variants are handled differently. We will solve the DOMINATING SET variant of each problem on this graph to give an intuitive understand of how each problem differs from the others.

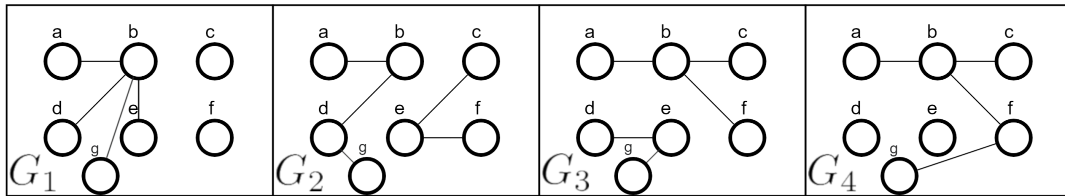

Figure 4.1: A temporal graph $\mathcal{G}$ with lifetime $\tau = 4$ and $n = 7$. We will solve every temporal domination variant on this graph.

4.1 Domination Rules

In each of the definitions for the sdf (3.3), rdf (3.4) and wdf (3.6), it is stated how a vertex u *dominates* another vertex v. We call a description for how a vertex can dominate another vertex a *domination rule.* There are three rules, R_{DOM}, R_{ROM} and R_{WEAK}, relating to the corresponding problems DOMINATING SET, ROMAN DOMINATION and WEAK ROMAN DOMINATION respectively.

Definition 4.1. *Let $G = (V, E)$ be a static graph, $v \in V$ a vertex and $f : V \to \mathbb{N}_0$ a domination function. A vertex $u \in V$ **dominates** v **in** f **according to domination rule***

> R_{DOM} *if $u \in N[v]$ and $f(u) \geq 1$.*
>
> R_{ROM} *if $u = v$ and $f(v) > 0$ or $u \in N(v)$ and $f(u) \geq 2$.*
>
> R_{WEAK} *if $u \in N[v]$ and $f(u) > 0$ and all vertices defended by f are also defended by $f_{u \to v}$.*

*Let R be that domination rule. We then also say that f **dominates** v **according to** R. If f dominates every vertex $v \in V$ according to R, then f is an R-**domination function for** G.*

A domination rule describes how a vertex can be dominated in a static graph. We will use them to describe how vertex appearances can be dominated. We will sometimes refer to DOMINATING SET, ROMAN DOMINATION and WEAK ROMAN DOMINATION as R_{DOM}-DOMINATION, R_{ROM}-DOMINATION and R_{WEAK}-DOMINATION respectively.

Casteigts and Flocchini suggest three ways DOMINATING SET can be translated to a temporal variant [8]. Akrida et al. [1] suggest another way to do this. These ways of translating DOMINATING SET to the temporal case can also be applied to ROMAN DOMINATION and WEAK ROMAN DOMINATION. Instead of giving each possible temporal problem, we will describe some temporal domination requirements that can be applied to any domination rule R.

4.2 TEMPORARY DOMINATION

We will avoid using the name given by Casteigts and Flocchini to the first temporal domination requirement (which was specifically applied to DOMINATING SET), which they called TEMPORAL DOMINATING SET [8]. Instead, we will use the name TEMPORARY DOMINATION to avoid confusion with other temporal variants.

Definition 4.2. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $v \in V$ a vertex, $f : V \to \mathbb{N}_0$ a domination function and R a domination rule. f* ***temporarily dominates*** *v according to R if f dominates v according to R in G_t for some time step $t \in T$. If f temporarily dominates every vertex $u \in V$ according to R, we call f an* ***R-temporary function****.*

A domination function f needs to dominate each vertex *at least once* (in at least one time step) in order to qualify as an R-temporary function. When R is equal to R_{DOM}, R_{ROM} or R_{WEAK} we call f a temporary sdf, rdf or wdf respectively.

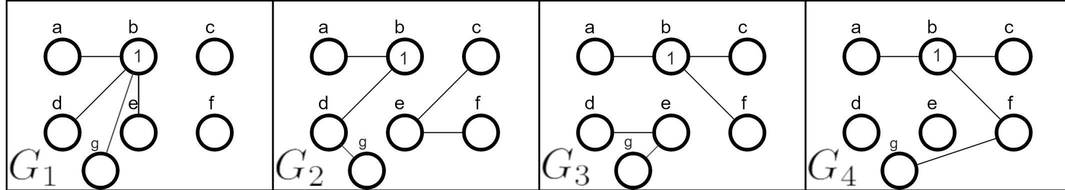

Figure 4.2: R_{DOM}-TEMPORARY DOMINATION solved on $\mathcal{G}$. We only need one vertex, since every vertex becomes the neighbour of b at least once.

Unsurprisingly, the corresponding TEMPORARY DOMINATION problems revolve around finding such a domination function of minimum size.

Problem 4.1. TEMPORARY DOMINATION*: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a domination rule R find the size of a minimum R-temporary function.*

These temporary problems are equivalent to solving the corresponding static problem on the underlying graph $G_\downarrow$ of the temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, as we will show in Section 5. For example, TEMPORARY DOMINATING SET (TEMPDOM) on $\mathcal{G}$ is equivalent to solving DOMINATING SET on $G_\downarrow$.

4.3 PERMANENT DOMINATION

The next temporal domination requirement, PERMANENT DOMINATION [8], is a more "strict" variant than TEMPORARY DOMINATION. Instead of dominating each vertex on at least one time step, we need to dominate it on every time step. This requirement requires us to define another type of domination.

Definition 4.3. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $v \in V$ a vertex, $f : V \to \mathbb{N}_0$ a domination function and R a domination rule. f* ***permanently dominates*** *v according to R if f dominates v according to R in G_t for every time step $t \in T$. If f permanently dominates every vertex $u \in V$ according to R, we call f an* ***R-permanent function****.*

A domination function needs to dominate each vertex on every time step in order to qualify as an R-permanent function, a "stricter" requirement than the temporary variant. We will also use the term permanent sdf, rdf or wdf depending on R.

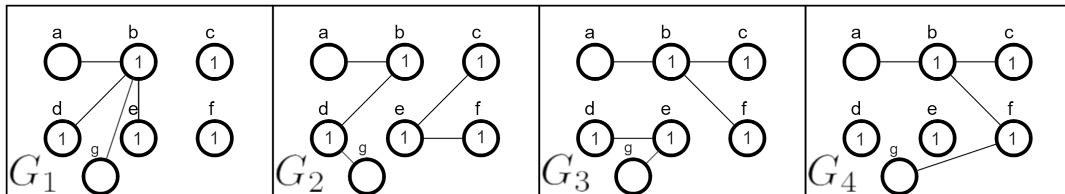

Figure 4.3: When we solve the permanent variant, we need at least 5 vertices because many of these vertices become isolated from all others at some point in the lifetime of $\mathcal{G}$.

Problem 4.2. PERMANENT DOMINATION*: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a domination rule R, find the size of a minimum R-permanent function.*

We solve the permanent variants of DOMINATING SET, ROMAN DOMINATION and WEAK ROMAN DOMINATION in Section 6 by creating dynamic programming algorithms based on algorithms for the static problems.

4.4 Periodic Domination

Akrida et al. suggest a more general temporal domination requirement called Periodic Domination [1] (PerDom). It asks us to dominate a vertex at least once every θ time steps, for some parameter θ. To formalize what this means, we need the concept of *windows*.

Definition 4.4. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and $\theta \in T$ an integer called the* ***period****. For $1 \leq t \leq \tau - \theta + 1$, let $G_t^\theta := (V, E_t \cup E_{t+1} \cup \cdots \cup E_{t+\theta-1})$ be the* ***t-th window*** *of $\mathcal{G}$. Let $N_t^\theta(v)$ and $N_t^\theta[v]$ be the open and closed neighbourhood of v respectively in the graph G_t^θ.*

Each window contains all the edges that exist at some point during the θ time steps that make up that window. To dominate a vertex in Periodic Dominating Set, it has to be dominated in each window.

Definition 4.5. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $v \in V$ a vertex, $f : V \to \mathbb{N}_0$ a domination function, R a domination rule and $\theta \in T$ the period of $\mathcal{G}$. f* ***periodically dominates*** *v according to R if f dominates v according to R in G_t^θ for every window $1 \leq t \leq \tau - \theta + 1$. If f periodically dominates every vertex $u \in V$ according to R, we call f an* ***R-periodic function****.*

We also refer to an R-periodic function as a periodic sdf, rdf or wdf depending on R.

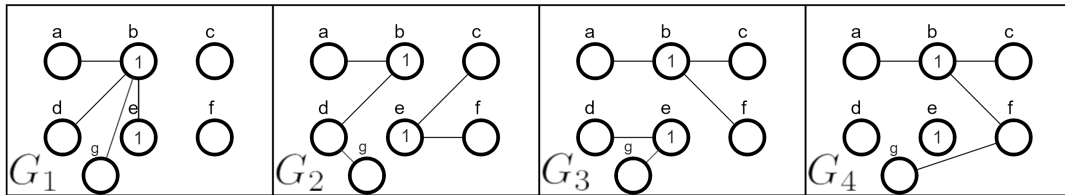

Figure 4.4: When the period is 2, the periodic variant requires each vertex to be dominated at least once every two time steps. On this graph, that means each vertex must be dominated once among G_1 and G_2, once among G_2 and G_3 and once among G_3 and G_4. We can do this using only 2 vertices.

This allows us to finally define Periodic Domination.

Problem 4.3. Periodic Domination*: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, a domination rule R and a period θ, find the size of a minimum R-periodic function.*

Both the temporary and permanent requirements are actually instances of the periodic variant. If $\theta = 1$, then the time windows are the same as the layers, and thus every vertex has to be dominated in every time step. This is equivalent to Permanent Domination. If $\theta = \tau$, then there is only one time window encompassing all the layers in which each vertex needs to be dominated. This is equivalent to Temporary Domination. Whenever $1 < \theta < \tau$ the order of the time steps becomes relevant to the solution of the problem, something that is not true for the former two variants. Some might say this makes periodic variants "more temporal" than the other problems, meaning that they use more of the properties of temporal graphs.

As it turns out, we can reduce any instance of Periodic Domination into an instance of Permanent Domination, allowing us to use the algorithms from Section 6 for these problems as well. Periodic variants will be discussed in Section 7.

4.5 Evolving Domination

The previous three temporal domination requirements have revolved around a single domination function that affects all the layers simultaneously. The next requirement, Evolving Domination [8], asks us to provide a domination function for each layer separately.

Problem 4.4. Evolving Domination*: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a domination rule R, find the total size $\sum_{t \in T} |f_t|$ of a sequence of domination functions $(f_1, \cdots, f_\tau)$ such that f_t dominates v according to R in G_t for every vertex $v \in V$ and every f_t is of minimum size.*

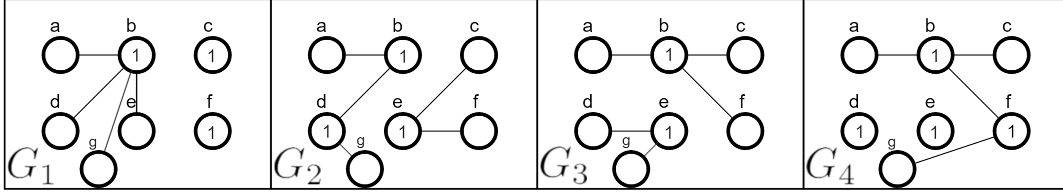

Figure 4.5: A separate minimum sdf for each layer, together forming an evolving domination.. The total number of vertex appearances we use, is 12.

All of the problems so far, except the evolving variants require a single domination function that dominates the entire temporal graph, according to the corresponding domination rule. Evolving variants stick out because these problems require a separate domination function for each layer. It turns out that we cannot solve evolving problems faster than simply performing a static Dominating Set, Roman Domination or Weak Roman Domination algorithm on each layer separately. We will discuss the evolving variants in Section 8.

4.6 k-Fold Domination

While the previous four requirements were from existing literature, the rest of this section will focus on requirements that, to the best of our knowledge, have not been studied before. k-Fold Domination is a new temporal domination requirement inspired by the temporary and permanent requirements. Much like the periodic requirement, k-Fold Domination is a generalisation of the former two. It asks for every vertex to be dominated in at least k layers. If $k = 1$, then k-Fold Domination is equivalent to Temporary Domination and if $k = \tau$ then it is equivalent to Permanent Domination.

Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph. For any vertex v, domination function $f : V \to \mathbb{N}_0$ and domination rule R, let $D_{vfR} = \{t \in T : v \text{ is dominated by } f \text{ according to } R \text{ in } G_t\}$ be the set of time steps in which v is dominated by f according to R.

Definition 4.6. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $v \in V$ a vertex, $f : V \to \mathbb{N}_0$ a domination function, R a domination rule and $k \leq \tau$ a positive integer. f **dominates** v k**-fold** according to R if $|D_{vfR}| \geq k$. If f dominates every vertex $u \in V$ k-fold according to R, we call f an R, k**-fold function**.*

We also refer to an R, k-fold function as a k-fold sdf, rdf or wdf depending on R.

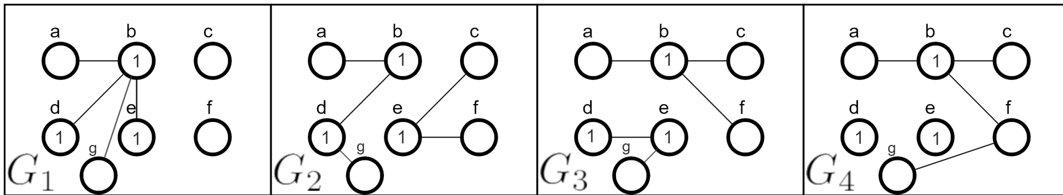

Figure 4.6: In R_{Dom},3-Fold Domination, each vertex must be dominated in at least 3 time steps. We achieve this by using 3 vertices.

This allows us to define the next problem.

Problem 4.5. **k-Fold Domination:** *Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, a domination rule R and a positive integer $k \leq \tau$, find the size of a minimum R, k-fold function.*

The methods used to solve the permanent variants of Dominating Set and Roman Domination can be applied to the k-fold variants of these problems as well. These solutions will be covered in Section 9.

Note that this requirement does not ask for any vertex to be dominated by k vertices, like the k-Tuple Dominating Set problem [18] [21]. That problem could be used to create another domination rule that can be combined with a temporal domination requirement, however.

4.7 Marching Domination

The military strategy that Roman Domination is based on revolves around the notion that armies are able to march to neighbouring cities to protect them in case of an attack. Marching Domination is based on that notion. In this problem, armies are capable of moving to other vertices during the transition between two time steps. By moving to different vertices, an army can defend a different part of the graph.

In the Marching Domination requirement we do not ask for a domination function. Instead, a solution to this problem is called a *strategy* and is represented by a list of *marches*.

Definition 4.7. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph. A* **march** *is a sequence $a = (a^1, \cdots, a^\tau)$ of τ vertices in V such that for each pair of consecutive vertices a^t, a^{t+1} in the sequence, we have that $a^{t+1} \in N_t[a^t]$. A* **strategy** *$s = \{a_1, \cdots, a_k\}$ is a multiset (a special type of set that allows duplicates: a_i and a_j are allowed to be equal even if $i \neq j$) of marches. The* **size** *of a strategy is the number $|s|$ of marches in s.*

Each march in a solution to Marching Domination represents the movements of one army. Therefore, the size of a strategy equals the number of armies used. A strategy is a multiset rather than a set because two armies could potentially walk the same route. The requirement that $a^{t+1} \in N_t[a^t]$ for each pair of consecutive vertices a^t, a^{t+1} in a march represents the fact that on each time step t, an army can either remain on the same vertex, or march over an edge $e \in E_t$ that exists on that time step.

For a strategy to be a valid solution to Marching Domination, each layer must be dominated by the armies. Let $s = \{a_1, \cdots, a_k\}$ be a strategy, then, on time step t, army i is stationed on vertex a_i^t. We can use this to deduce a domination function. First, for each vertex $v \in V$, let $\mathbb{I}_v$ be the indicator function such that

$$\mathbb{I}_v(u) = \begin{cases} 1 & \text{if } u = v \\ 0 & \text{otherwise} \end{cases}.$$

Then, the domination function $s_t : V \to \mathbb{N}_0$ for strategy s on time step t is defined as

$$s_t(v) = \sum_{a \in s} \mathbb{I}_v(a^t).$$

Definition 4.8. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and R a domination rule. A strategy s is an* **R-marching domination** *for $\mathcal{G}$ if for every time step t we have that s_t is an R-domination function for G_t. If the value of R is implicit, we simply call s a* **marching domination**.

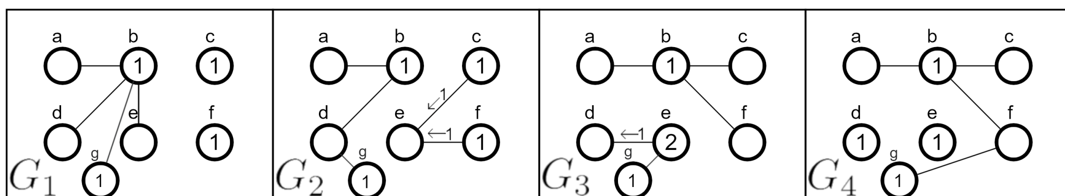

Figure 4.7: To achieve a minimum solution for the marching domination, we are required to station at least two armies to vertex e on the third turn, in order to get it to vertex d on the last time step, which is necessary because otherwise, d is not defended.

his allows us to finally define the Marching Domination problem.

Problem 4.6. Marching Domination: *Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a domination rule R, find the size of a minimum R-marching domination.*

Like with Periodic Domination, the solution to Marching Domination depends on the order of the time steps, making it more temporal than problems where the order does not matter. We cover Marching Domination in Section 10.

4.8 Temporal Defense-Like Domination

Finally, to adapt Defense-Like Domination to the temporal case, we will not use the temporary, permanent, periodic, evolving, k-fold and marching requirements used for Dominating Set, Roman Domination and Weak Roman Domination. Instead, we will organically convert it to the temporal case by making the number of turns k equal to the number of time steps τ and having each attack A_t take place on time step t. On turn t, the armies moved by the defender to protect against attack A_t can only use edges in G_t.

To bring back the analogy of a board game, in this temporal variant the game board is a temporal graph instead of a static one. The parameters retain the same meaning as before, though we drop the k and simply use τ to refer to the number of turns/time steps. Since τ is given by the temporal graph, we do not need to provide it again to define the problem. The rules are largely the same. The only difference is that on turn i, the defender can only move armies over edges that exist in layer G_t. This changes the definition of a partial, and valid move.

Definition 4.9. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, t a time step, $f : V \to \mathbb{N}_0$ a domination function on G_t and a, h, c and s the parameters. Let $g : V \to \mathbb{N}_0^2$ be an intermediate function on G_t. If the minimum number of edges on a path from u to v in G_t is $d^t_{u,v} \leq s$ and $g(u).x > 0$, then the intermediate function $g_{u\to v}$ is called a* ***partial move*** *for g on time step t. Now let $g_0, \cdots, g_h$ be a sequence of intermediate functions on G_t such that $g_0(v).x = f(v)$ for every vertex $v \in V$ and g_i is a partial move for g_{i-1} on time step t for $1 \leq i \leq h$. Let $f'(v) = g_h(v).x + g_h(v).y$ for all vertices $v \in V$. If $f'(v) \leq c$ for all vertices $v \in V$, then f' is a* ***valid move*** *for f on time step t.*

The definition of defending against an attack remains the same as for the non-temporal variant of this problem. For an initial domination function to defend against an attack sequence, there must be a sequence of τ valid moves starting with the initial domination function that each defend against the attack of the corresponding time step.

Definition 4.10. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $f : V \to \mathbb{N}_0$ a domination function, $A = (A_1, \cdots, A_\tau)$ an attack sequence and a, h, c and s the parameters. Initial domination function f* ***defends against*** *attack sequence A if there exists a sequence $f_0, \cdots, f_\tau$ of domination functions such that $f_0 = f$ and f_t is a valid move for f_{t-1} on time step t that defends against attack A_t for every time step t.*

With that, the *temporal defense-like domination* follows naturally:

Definition 4.11. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, and a, h, c and s the parameters. A domination function $f : V \to \mathbb{N}_0$ that defends against any attack sequence is called a* ***temporal defense-like domination*** *(tdld). The* ***size*** *of a tdld is the sum of the labels $\sum_{v\in V} f(v)$.*

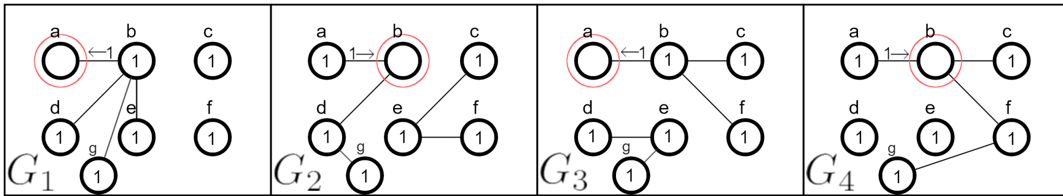

Figure 4.8: For $a = 1$, $h = 1$, $c = \infty$ and $s = 1$, we need to put an army on every vertex, with one exception: since a and b are always connected, we only need one army between the two of them. The attacker will not be able to win, because if they attack a vertex without an army, the army can always protect the attacked vertex without leaving the other vertex defenseless. c and f start as isolated vertices, forcing the defender to put armies on them, since leaving them without an army would allow the attacker to win immediately by attacking one of them. d, g and e also need to have armies, because on the first time step, we can only defend them using the army on b, and if that army moves, a can be attacked in time step 2. Note that a tdld is not a sequence of domination functions, only the initial domination function matters. It must be able to defend against any attack sequence through a sequence of valid moves.

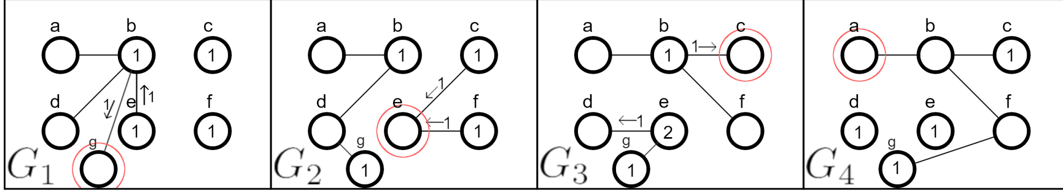

Figure 4.9: If we use $h = \infty$ (we may move any number of armies on one turn, instead of just one), but otherwise keep the same parameters, four armies are not enough. On the third time step, the set $\{d, e, g\}$ must contain three armies in total. If at least two are not present on $\{d, e, g\}$ by time step 3, there is no way for d and e to each have an army on time step 4. The third army is needed because if g is attacked on time step 3, we are forced to send an army to it, which means that d and e cannot both have an army by time step 4. In addition, at the start of third time step, there must be an army on b, otherwise either a, c or f can be attacked without an army being close enough to defend it. Therefore, we know that if we use four armies, then on time step 3 a, c and f do not have an army, allowing the attacker to force the defender to send an army away from b, which in turn allows the attacker to win by attacking whichever neighbour of b is undefended by time step 4.

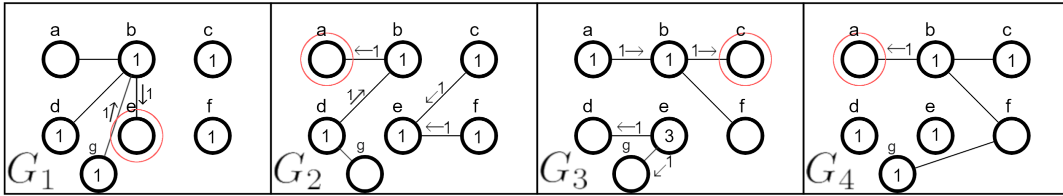

Figure 4.10: With $h = \infty$, 5 armies is enough to dominate the graph. Whatever the attacker does, we can make sure to have one vertex on b, three vertices on $\{d, e, g\}$ and one vertex somewhere on $\{a, b, c, f\}$ by time step 3. That is enough to guarantee a victory for the defensive player.

This finally leads us to the temporal variant of Defense-Like Domination.

Problem 4.7. Temporal Defense-Like Domination: *Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and parameters a, h, c and s, find a tdld of minimum size that satisfies the constraints imposed by the parameters.*

It is also possible to define temporary, permanent, periodic, evolving, k-fold and marching variants for Defense-Like Domination. The permanent variant, for example, would ask for a domination function that is a dld on every layer. In that problem, the "game" does not take place over the given time steps $1, \cdots, \tau$, with each attack happening on a different time step. Rather, every time step t sees its own complete game played out on the static graph that is the layer G_t. This is a rather convoluted way to approach Defense-Like Domination, especially since the static problem so naturally lends itself to a temporal application. Our temporal interpretation of the problem also highly relies on the order that the layers appear in, like in Periodic Domination and Marching Domination.

Part II

Temporal Domination Algorithms

Now that the theory of temporal graphs and domination problems has been provided, and several temporal domination requirements have been defined in the first part, this second part of the thesis will focus on providing algorithms that solve some of these temporal domination problems. Speed is our priority in finding these algorithms. Therefore our algorithms may require exponential space. We use a variety of methods depending on what is necessary to solve the problem at hand. For example, we use parameterized algorithms parameterized by the treewidth of the underlying graph to solve the permanent, periodic and k-fold variants of Dominating Set and Roman Domination, but we use exponential time algorithms for the marching variants.

Whenever a solution depends almost entirely on an existing algorithm, we will only provide the running time as well as a reference to that algorithm. Whenever a solution is original or requires large portions of the existing algorithm to be changed, we will provide the entire algorithm in pseudocode.

A summarized list of the running times that we achieve in this part will be provided in the third part of the thesis.

5 Temporary Domination

In this section we view Temporary Domination as defined in Section 4.2. Depending on the domination rule, there are three different variants of the problem that we will cover. We will redefine the problem for each domination rule.

Problem 5.1. Temporary Dominating Set: *Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, find the size of a minimum temporary sdf.*

Problem 5.2. Temporary Roman Domination: *Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, find the size of a minimum temporary rdf.*

Problem 5.3. Temporary Weak Roman Domination: *Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, find the size of a minimum temporary wdf.*

Temporary domination problems ask for a domination function that dominates each vertex at least once in the lifetime of a temporal graph. These problems are actually quite trivial if we have an algorithm that solves the static version of the domination problem in question.

Theorem 5.1. TempDom, TempRom *and* TempWeak *on a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ are equivalent to* Dominating Set, Roman Domination *and* Weak Roman Domination *on $G_\downarrow$ respectively.*

We will now prove this theorem for Temporary Dominating Set only. The proofs for Temporary Roman Domination and Temporary Weak Roman Domination are almost identical to this proof and only need a slight change in wording. Therefore these proofs will be left to the reader.

Proof of Theorem 5.1. Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, let $v \in V$ be a vertex in $\mathcal{G}$ and let $f : V \to \mathbb{N}_0$ be a domination function. If $f(v) = 1$, then f dominates v in $G_\downarrow$ and temporarily dominates v in $\mathcal{G}$. If $f(v) = 0$, then v is temporarily dominated by f if there is at least one edge appearance $e^t \in E^t$ that connects v^t to a vertex appearance u^t with $f(u) = 1$. Every edge of the underlying graph appears at least once in $\mathcal{G}$ and any edge that appears in $\mathcal{G}$ is also an edge in $G_\downarrow$. Therefore v is temporarily dominated by f in $\mathcal{G}$ if and only if v is dominated by u in $G_\downarrow$. Therefore a domination function f is a temporary sdf on $\mathcal{G}$ if and only if it is an sdf on $G_\downarrow$. Since the collection of temporary sdf's on $\mathcal{G}$ is equal to the collection of sdf's on $G_\downarrow$, the minimum size of an sdf on $G_\downarrow$ is also the minimum size of a temporary sdf on $\mathcal{G}$ and vice versa. Therefore solving one of those problems is enough to solve both of them. □

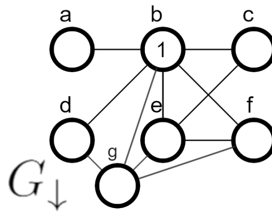

Figure 5.1: The temporary domination used in Figure 4.2 corresponds to this sdf on the underlying graph $G_\downarrow$ of $\mathcal{G}$.

Since the problems are equivalent to the static problems on which they are based on a static graph of the same size, we can solve this problem using any algorithm A for the corresponding static problem.

To get optimal running times for our three temporal problems, we will use the currently fastest algorithms for Dominating Set and (Weak) Roman Domination. For Dominating Set, this is the algorithm by Iwata [23] that uses a method called the *potential method*. He provides two algorithms in this paper. The first runs in $\mathcal{O}(1.4864^n)$ time and uses polynomial space. The second requires $\mathcal{O}(1.4689^n)$ time and space. Since we are focusing on speed alone in this thesis, we will use the second algorithm.

Theorem 5.2. Temporary Dominating Set *can be solved in $\mathcal{O}(1.4689^n)$ and space.*

For Roman Domination, this is the algorithm by Shi and Koh [28]. They provide two algorithms. The first runs in $\mathcal{O}(1.5673^n)$ and polynomial space and the second runs in $\mathcal{O}(1.5014^n)$ and exponential space. Again, for this example, we will use the faster algorithm.

Theorem 5.3. Temporary Roman Domination *can be solved in* $\mathcal{O}(1.5014^n)$ *and exponential space.*

For Weak Roman Domination, this is the algorithm by Chapelle et al. [10]. They provide two algorithms. The first runs in $\mathcal{O}^*(2.2279^n)$ time and polynomial space and the second runs in $\mathcal{O}^*(2^n)$ time and exponential space. Again, we use the faster algorithm.

Theorem 5.4. Temporary Weak Roman Domination *can be solved in* $\mathcal{O}^*(2^n)$ *and exponential space.*[1]

6 Permanent Domination

Permanent domination problems ask for domination of each vertex *on every time step.* The technique that we used to solve Temporary Domination by applying a static algorithm to the underlying graph, does not work for Permanent Domination. The reason for this is that the underlying graph loses information about when a vertex is *not* dominated, which is not important in the temporary variants, but is crucial in solving the permanent ones. Instead, the problems have to be solved in more complicated ways. In the algorithms we found, τ plays a role in the time complexity, as it will in all other temporal domination problems from this point forward.

6.1 Parameterized Algorithm Structure

The algorithms for Permanent Dominating Set and Permanent Roman Domination are very similar. The algorithms are closely related to the algorithm given in the proof of Theorem 13 in [2], the algorithm given in Section 11.3 of [31] and the algorithm in Section 6 of [32]. We will use much of the same terminology used in those sources. The algorithms share the same structure, which will be explained here, and they differ only in how certain calculations are made.

Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph with timeline T and $\mathcal{X} = (\{X_1, \cdots, X_k\}, X)$ a nice tree decomposition of the underlying graph $G_\downarrow$ of width ℓ. For each node i of $\mathcal{X}$, we will define the *history vertex set* V_i. This is the set of all vertices that have appeared so far in X_i and all bags of the decendants of i.

Definition 6.1. *Let* $\mathcal{G} = (V, E, \tau, \lambda)$ *be a temporal graph,* $\mathcal{X} = (\{X_1, \cdots, X_k\}, X)$ *a nice tree decomposition of the underlying graph* $G_\downarrow$ *and* i *a node of* $\mathcal{X}$ *with children* K_i*. The* ***history vertex set*** *of node* i *is a vertex set* $V_i := \left(\bigcup_{j \in K_i} V_j\right) \cup X_i$.

[1] The $\mathcal{O}^*$ notation here means that there may be another polynomial factor.

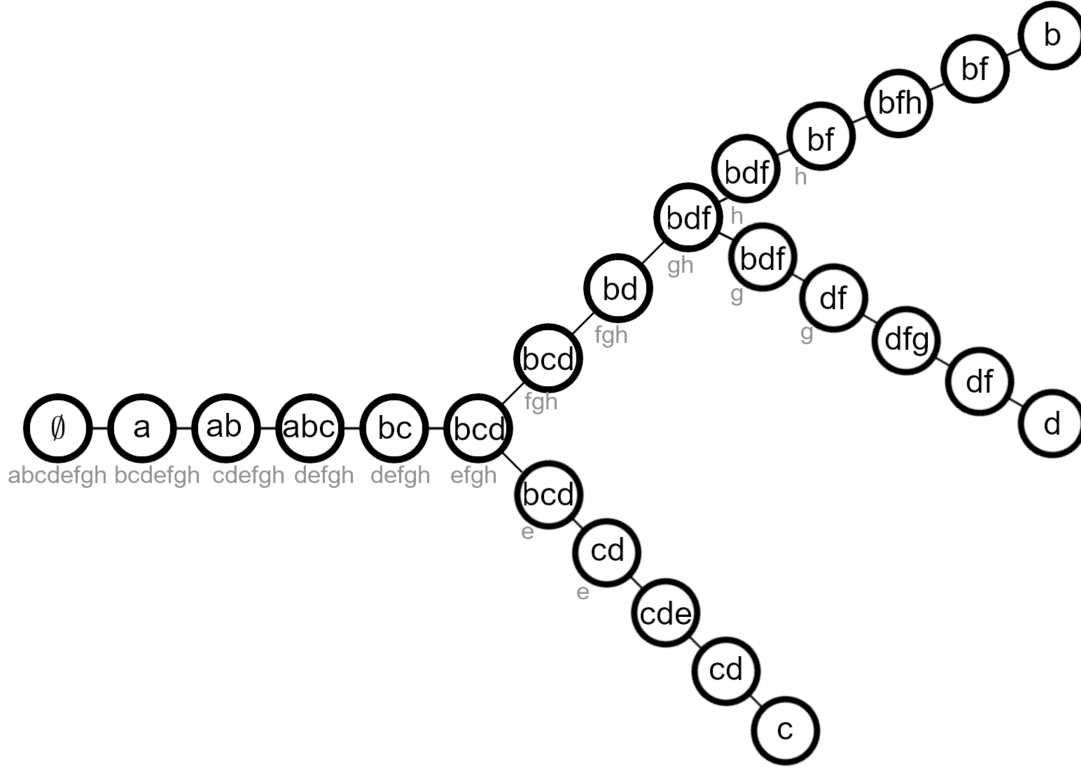

Figure 6.1: The nice tree decomposition of the graph from Figures 2.3 and 2.4, now showing the forgotten vertices underneath each node in gray. The history vertex set of each node is the union of the vertices shown in black and those shown in gray.

Note that for the root node r, we have that $V_r = V$. Every node i in $\mathcal{X}$ contains a table A_i. We will use the table A_i to count the number of *partial solutions* on node i for each possible *temporal colouring* of the vertices in X_i and each possible *solution size* κ. To understand how exactly the table entries will be used, we need to explain what partial solutions and temporal colourings are.

6.1.1 Partial Solutions

Whenever we refer to the domination of a vertex or a vertex set in this section, we will mention what domination rule we are using only if this matters in the current context. In many instances, the domination rule can be any of the three mentioned. In other moments, it will be clear from the context which domination rule we use, for example, in Section 6.2 we only talk about the domination rule R_{DOM}.

A partial solution is a domination function on V_i. It is not necessary for a partial solution to dominate all vertices in V_i but it has to dominate V_i/X_i. In other words, the vertices in X_i are the only ones that do not have to be dominated.

Definition 6.2. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X} = (\{X_1, \cdots, X_k\}, X)$ a nice tree decomposition of the underlying graph $G_\downarrow$ and i a node of $\mathcal{X}$. A* ***partial solution*** *of node i is a domination function $f : V_i \to \mathbb{N}_0$ such that f permanently dominates every vertex $v \in V_i/X_i$ in $\mathcal{G}[V_i]$.*

A partial solution is similar, but not necessarily equal to a *permanent domination function* as described in Section 4.3. It has to permanently dominate all vertices of $\mathcal{G}[V_i]$ that are in V_i/X_i and can also use vertices from X_i to do so, but is not required to dominate the vertices in X_i itself.

Consider the following temporal graph $\mathcal{G}$.

Figure 6.2: A temporal graph $\mathcal{G}$ with $\tau = 2$. Its underlying graph, shown in the leftmost picture, is the static graph from Figure 2.3. The middle picture displays the first layer, and the final picture the second.

We can use the same nice tree decomposition from Figure 6.1. To illustrate partial solutions for R_{ROM}-PERMANENT DOMINATION, we provide an example on two nodes in particular.

Figure 6.3: We focus on a forget node and its child, a join node.

A partial solution on the join node may look like this.

Figure 6.4: The subgraph induced by the history vertex of the join node with the forgotten vertices and the corresponding edges in grey. The domination function displayed is a valid partial solution for this node. All grey vertices are dominated. The black vertices are not, because they do not have to be.

Once we forget d in the next node, it also needs to be dominated.

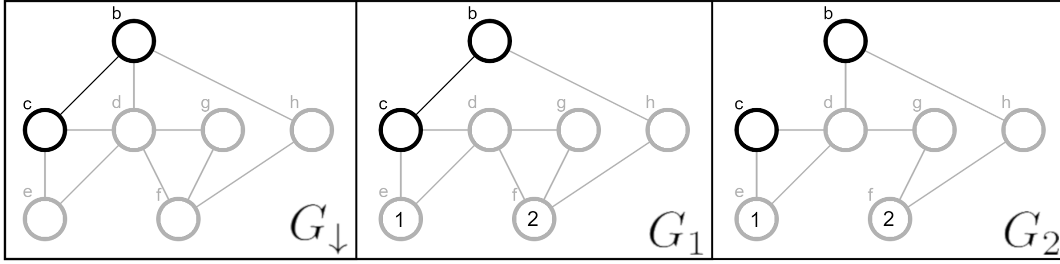

Figure 6.5: The domination function from Figure 6.4 is no longer a valid partial solution, because d is not permanently dominated.

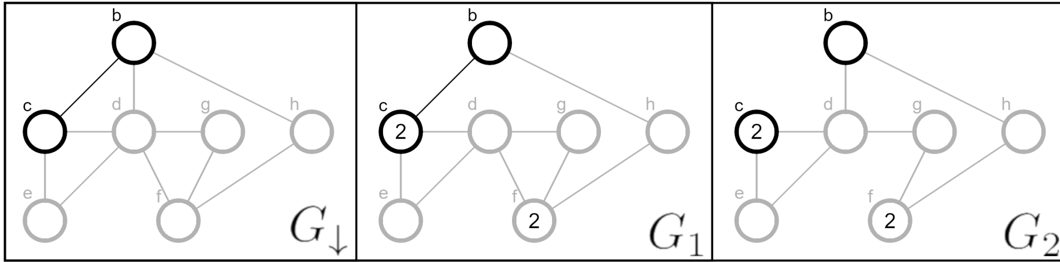

Figure 6.6: The domination function shown here is a valid partial solution for both this node and the previous node.

6.1.2 Temporal Colourings

A temporal colouring for bag X_i assigns a "colour" to each vertex in X_i for each time step. There are five colours: 2, 1, ?, 0 and $\hat{0}$.

Definition 6.3. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X} = (\{X_1, \cdots, X_k\}, X)$ a nice tree decomposition of the underlying graph $G_\downarrow$ and i a node of $\mathcal{X}$. Let $X_i = (x_{i_1}, \cdots, x_{i_{n_i}})$. A **temporal colour** for vertex x_{i_j} is a vector $c_j = \left(c_j^1, \cdots, c_j^\tau\right) \in \{2\}^\tau \cup \{1\}^\tau \cup \left\{?, 0, \hat{0}\right\}^\tau$ that assigns a colour c_j^t to each vertex appearance $x_{i_j}^t$. A **temporal colouring** of bag $X_i = (x_{i_1}, \cdots, x_{i_{n_i}})$ is a vector $c = (c_1, \cdots, c_{n_i}) \in \left(\{2\}^\tau \cup \{1\}^\tau \cup \left\{?, 0, \hat{0}\right\}^\tau\right)^{n_i}$ that assigns a temporal colour c_j to each vertex $x_{i_j} \in X_i$.*

A temporal colouring c for bag X_i assigns to every vertex $x_{i_j} \in X_i$ a *temporal colour* c_j which, in turn, is a vector that assigns colour $c_{j_t}^t$ to vertex appearance $x_{i_j}^t$ of vertex x_{i_j} at time step t. A vertex x_{i_j} can be coloured differently on two different time steps. However, a vertex can not be partially coloured 2 or 1, since f is a function that remains the same every time step. Either $c_j = (2, \cdots, 2)$ (a vector with τ coordinates that are all equal to 2), $c_j = (1, \cdots, 1)$ or $c_j \in \left\{?, 0, \hat{0}\right\}^\tau$. If all coordinates of c_j are equal to the same colour $C \in \left\{2, 1, ?, 0, \hat{0}\right\}$, we simply write $c_j = C$.

Definition 6.4. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X} = (\{X_1, \cdots, X_k\}, X)$ a nice tree decomposition of the underlying graph $G_\downarrow$, f a partial solution for node i of $\mathcal{X}$ and $R \in \{R_{\text{DOM}}, R_{\text{ROM}}\}$ a domination rule. f **satisfies** temporal colouring c if for each vertex $x_{i_j} \in X_i$ we have that:*

$$f(x_{i_j}) = \begin{cases} 2 & \text{if } c_j = 2 \\ 1 & \text{if } c_j = 1 \\ 0 & \text{if } c_j \in \left\{?, 0, \hat{0}\right\}^\tau \end{cases}$$

and for each vertex $x_{i_j} \in X_i$ and time step $t \in T$ we have:

if $c_j^t = \hat{0}$, then for every neighbouring vertex $y \in N_t(x_{i_j}) \cap V_i$ we have $f(y) \neq 1$ if $R = R_{\text{DOM}}$ and $f(y) \neq 2$ if $R = R_{\text{ROM}}$.

if $c_j^t = 0$, then there is a neighbouring vertex $y \in N_t(x_{i_j}) \cap V_i$ such that $f(y) = 1$ if $R = R_{\text{DOM}}$ and $f(y) = 2$ if $R = R_{\text{ROM}}$.

For any partial solution f that *satisfies* c, the colour given to vertex appearance v^t represent the following options:

- 2: $f(v) = 2$.
- 1: $f(v) = 1$.
- ?: $f(v) = 0$ and we do not know whether v is dominated in $G_t[V_i]$.
- 0: $f(v) = 0$ and v is dominated in $G_t[V_i]$.
- $\hat{0}$: $f(v) = 0$ and v is not dominated in $G_t[V_i]$.

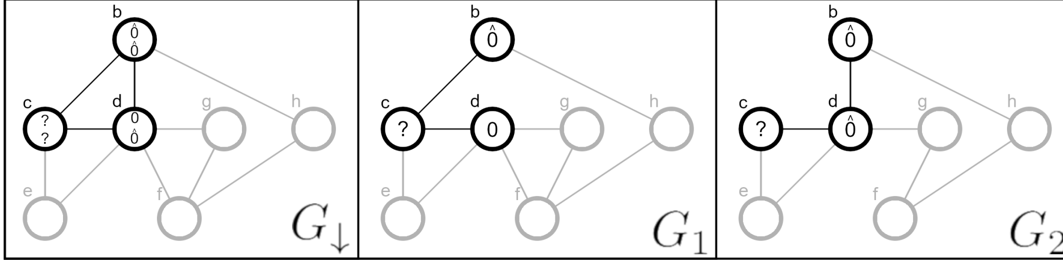

Figure 6.7: The partial solution from Figure 6.4 satisfies this colouring.

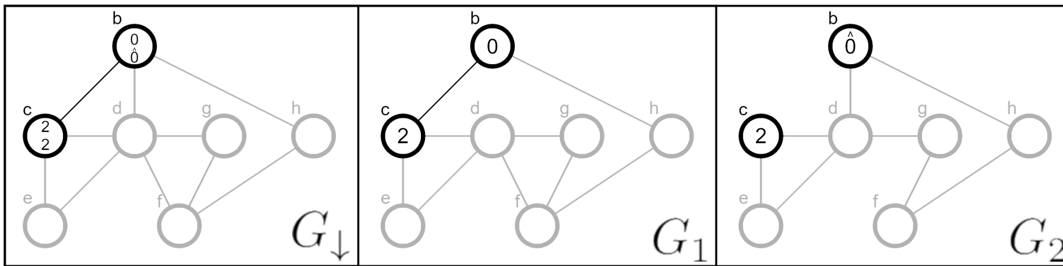

Figure 6.8: The partial solution from Figure 6.6 satisfies this colouring.

The colours ?, 0 and $\hat{0}$ for vertex appearance v^t all correspond with $f(v) = 0$, but they have different meanings. Vertex appearances x_{i_j} in X_i with $c_j =?$ *might* be dominated by f in V_i. We pretend we do not know this for certain, even if we can clearly see they have a neighbouring vertex appearance in X_i that is coloured 1 in the Permanent Dominating Set problem, or 2 in the Permanent Roman Domination problem. Vertex appearances x_{i_j} in X_i with $f(x_{i_j}) = 0$ represent vertex appearances that are already dominated by f. Even if they do not have a neighbour in X_i that dominates it, they might be dominated by vertices in V_i/X_i, i.e. vertices that have already been forgotten. Vertex appearances x_{i_j} in X_i with $f(x_{i_j}) = \hat{0}$ are not yet dominated. However it is possible that they have a neighbouring vertex appearance in X_i that is coloured in such a way that any partial solution that satisfies this temporal colouring would dominate it. In this case, we call c *locally invalid.*

Definition 6.5. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, R a domination rule, $\mathcal{X} = (\{X_1, \cdots, X_k\}, X)$ a nice tree decomposition of the underlying graph $G_\downarrow$, i a node of $\mathcal{X}$ and $X_i = (x_{i_1}, \cdots, x_{i_{n_i}})$ the bag of node i. A temporal colouring $c = (c_1, \cdots, c_{n_i}) \in \left(\{2\}^\tau \cup \{1\}^\tau \cup \{?, 0, \hat{0}\}^\tau\right)^{n_i}$ is **locally invalid** if, for some pair of neighbouring vertex appearances $\left\{x^t_{i_j}, x^t_{i_k}\right\} \in E^t$, we have that $c^t_j = \hat{0}$ and $c^t_k = 1$ in case $R = R_{\mathrm{DOM}}$ and $c^t_k = 2$ in the case $R = R_{\mathrm{ROM}}$.*

It is trivial to see that for any locally invalid temporal colouring c, there can be no partial solution f that satisfies c. It is important we understand how and where locally invalid temporal colourings can occur, so that we can make sure our algorithm handles them correctly. Leaf nodes cannot have locally invalid temporal colourings, and when we handle forget nodes and join nodes, we can easily avoid locally invalid temporal colourings by only building on the valid temporal colourings of the children of these nodes.

Lemma 6.1. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of the underlying graph $G_\downarrow$ and i a leaf node of $\mathcal{X}$. Then any temporal coloring c on i is not locally invalid.*

Proof of Lemma 6.1. Since i only contains one vertex x, it cannot have a neighbour, thus the situation where a coloring is locally invalid cannot occur.

□

Lemma 6.2. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of the underlying graph $G_\downarrow$, i a forget node of $\mathcal{X}$ with child j and c a temporal coloring on j that is not locally invalid. Let the temporal coloring c' on i be c limited to vertices in X_i. Then c' is not locally invalid.*

Proof of Lemma 6.2. Since c is not locally invalid, we know that for each pair of neighbouring vertex appearances $\left\{x^t_{j_a}, x^t_{j_b}\right\} \in E^t$ for $x_{j_a}, x_{j_b} \in X_j$, we have that if $c^t_a = 1$ or $c^t_a = 2$ then $c^t_b \neq \hat{0}$. Since X_i only contains vertices that are also in X_j, and c' gives the same color to them in X_i as c does in X_j, no local invalidity can occur.

□

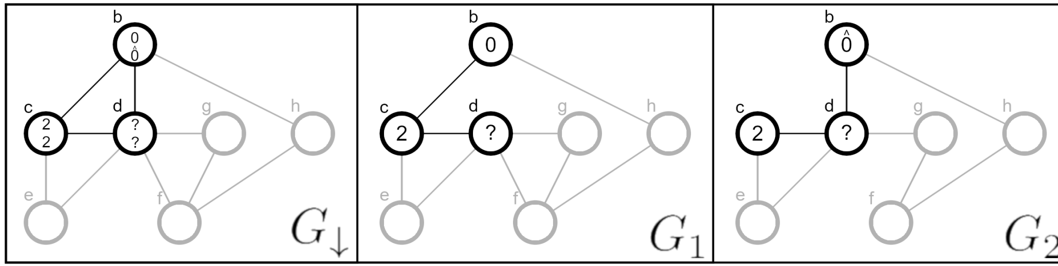

Figure 6.9: A temporal colouring on the join node highlighted in Figure 6.3. This temporal colouring satisfies the partial solution in Figure 6.4. The temporal colouring that naturally follows from this by removing d is displayed in Figure 6.8. Both temporal colourings are not locally invalid.

Lemma 6.3. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of the underlying graph $G_\downarrow$, i a join node of $\mathcal{X}$ with children j and k and c a temporal coloring on both j and k that is not locally invalid on either of them. Then c is not locally invalid on i.*

Proof of Lemma 6.3. Since c is not locally invalid on either j or k, it can not be locally invalid on i, because $X_i = X_j = X_k$.

□

These three lemmas together imply that, because the way our algorithms handle these nodes, locally invalid temporal colourings only have to be taken into account when handling introduce nodes.

Going back to the previous example, consider the following introduce node, whose child is the forget node we previously covered.

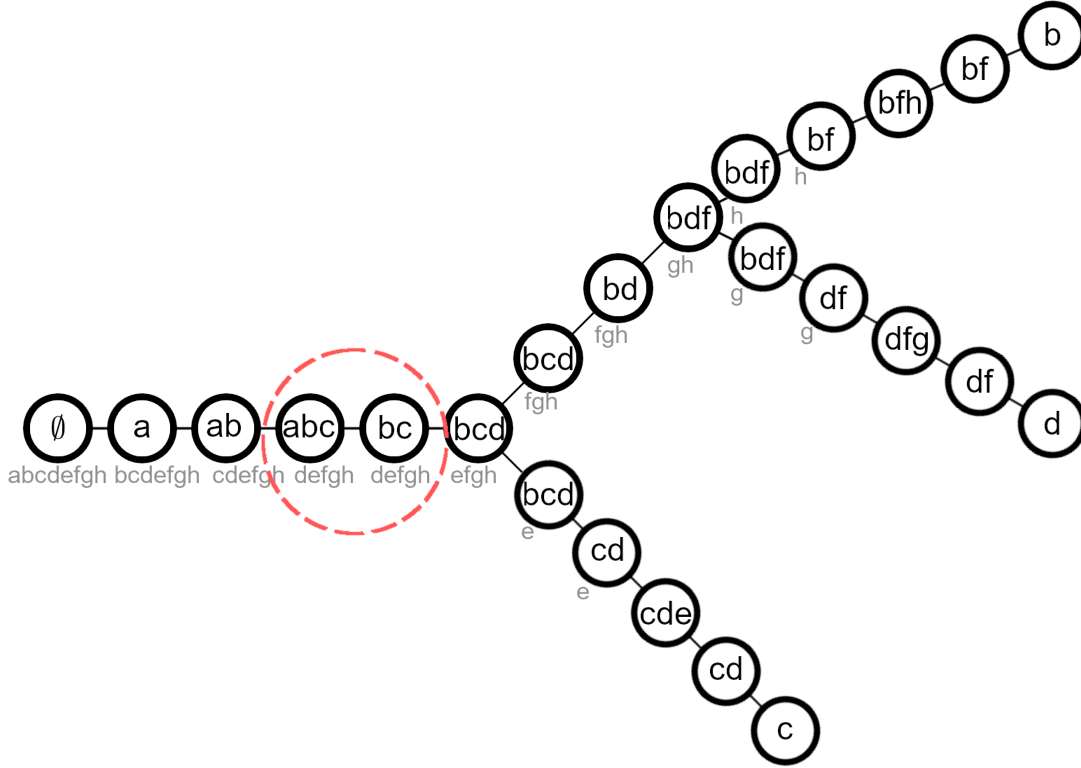

Figure 6.10: The highlighted introduce node introduces a to the bag.

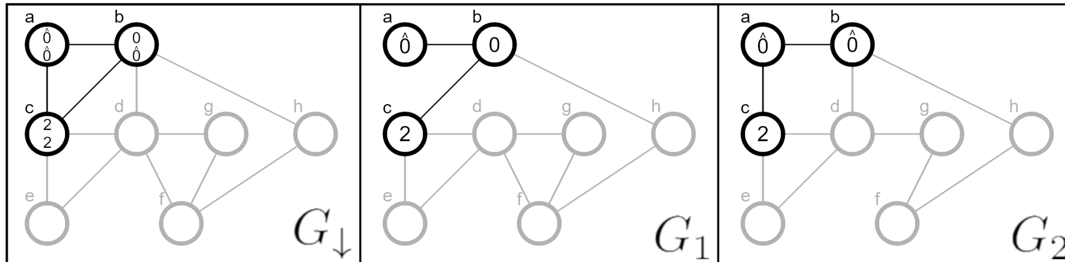

Figure 6.11: If we extend the temporal colouring on the child of the introduce node from Figure 6.8, to a temporal colouring on the introduce node which includes a, there is a possibility that we get a locally invalid temporal colouring, since the colors for a and c create a conflict in time step 2.

6.1.3 Dynamic Programming

Each node i of the nice tree decomposition $\mathcal{X}$ contains a table A_i. Table entry $A_i(c, \kappa)$ stores the number of partial solutions of size κ on i that satisfy temporal colouring c. We visit the nodes of $\mathcal{X}$ from the leaves to the root in a BFS manner, filling in the tables of the nodes as we encounter them. Since $\mathcal{X}$ is a nice tree decomposition, we only have to take four types of nodes into account, the leaf nodes (which each contain only one vertex), forget nodes, introduce nodes and join nodes. We fill the table of each introduce, forget or join node by referring to table entries of the child nodes. The way these nodes are handled differs between the problems PERMANENT DOMINATING SET and PERMANENT ROMAN DOMINATION.

In each introduce node, we need to make sure that any locally invalid temporal colourings are not taken into account. In each forget node, we need to make sure that we only take into account the temporal colours 2 (though only in the PERMANENT ROMAN DOMINATION problem), 1 and 0 for the forgotten vertex. These three temporal colours are the only temporal colours for the forgotten vertex that will lead to an acceptable final solution, as they represent the situations where the vertex is permanently dominated.

Once we reach the root node r of $\mathcal{X}$ and filled its table A_r, we can terminate the algorithm. The root node has an empty bag, thus we do not need to check any temporal colourings. We look for the minimum value of κ for which $A_r(\emptyset, \kappa) > 0$. This is the minimum size of an R-permanent domination function.

6.1.4 Reducing the Table

We can make the algorithm faster by ignoring the colour 0. We can fill the table entries for the temporal colourings that use only the colours 2, 1, ? and $\hat{0}$, which saves time over also using the colour 0. Then, when we reach a forget node where vertex x is forgotten, we can calculate the table entries where x is coloured 0 by using techniques based on *fast subset convolution* and the *covering product* [3] [30] [31]. Our treatment is along the lines of state changes as inspired by van Rooij [31].

Let us define a partial ordering $\prec$ on the colours. Let $C \prec C$ for every colour $C \in \{2, 1, ?, 0, \hat{0}\}$ and let $0 \prec ?$ and $\hat{0} \prec ?$. This partial ordering can then be extended to temporal colours. Let i be a node of $\mathcal{X}$ and $c_j = (c_j^1, \cdots, c_j^\tau)$ and $\hat{c}_j = (\hat{c}_j^1, \cdots, \hat{c}_j^\tau)$ two temporal colours for some vertex $x_{i_j} \in X_i$. We say that $\hat{c}_j \prec c_j$ if $\hat{c}_j^t \prec c_j^t$ for all time steps $t \in T$. This can once again be extended to temporal colourings. Let $c = (c_1, \cdots, c_n)$ and $\hat{c} = (\hat{c}_1, \cdots, \hat{c}_n)$ be two temporal colourings on i, then $\hat{c} \prec c$ if $\hat{c}_j \prec c_j$ for all $1 \leq j \leq n$. This ordening represents the fact that partial solutions that satisfy $\hat{c}$ also satisfy c.

Lemma 6.4. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition for $G_\downarrow$, i a node of $\mathcal{X}$ and $R \in \{R_{\text{DOM}}, R_{\text{ROM}}\}$. Let $c = (c_1, \cdots, c_n)$ and $\hat{c} = (\hat{c}_1, \cdots, \hat{c}_n)$ be two temporal colourings on i such that $\hat{c} \prec c$ and f a partial solution on i that satisfies $\hat{c}$. Then f also satisfies c.*

Proof of Lemma 6.4. We will prove this using contraposition. Assume f is a partial solution that does not satisfy c, then we will prove it does not satisfy $\hat{c}$ either.

Case 1: There is a vertex $x \in X_i$ which is assigned temporal colour c_x in c such that:

$$f(x) \neq \begin{cases} 2 & \text{if } c_x = 2 \\ 1 & \text{if } c_x = 1 \\ 0 & \text{if } c_x \in \{?, 0, \hat{0}\}^\tau \end{cases}$$

In this case f does not satisfy $\hat{c}$ either.

Case 2: There is a vertex appearance $x^t \in V^t$ for $x \in X_i$ which is assigned colour $c_x^t = \hat{0}$ in c, and there is a neighbouring vertex $y \in N_t(x_{i_j}) \cap V_i$ such that $f(y) = 1$ in the case where $R = R_{\text{DOM}}$ and $f(y) = 2$ in the case where $R = R_{\text{ROM}}$. In this case f does not satisfy $\hat{c}$ either since $\hat{c} \prec c$ implies that $\hat{c}_x^t = \hat{0}$ too.

Case 3: There is a vertex appearance $x^t \in V^t$ for $x \in X_i$ which is assigned colour $c_x^t = 0$ in c, and for every neighbouring vertex $y \in N_t(x_{i_j}) \cap V_i$ we have $f(y) \neq 1$ in the case where $R = R_{\text{DOM}}$ and $f(y) \neq 2$ in the case where $R = R_{\text{ROM}}$. In this case f does not satisfy $\hat{c}$ either since $\hat{c} \prec c$ implies that $\hat{c}_x^t = 0$ too.

Therefore any partial solution f on i that satisfies $\hat{c}$ must also satisfy c.

□

Let $B_j(c, \kappa)$ be the set of partial solutions of size κ that satisfy temporal colouring c. Then $\hat{c} \prec c$ implies that $B_i(\hat{c}, \kappa) \subset B_i(c, \kappa)$. This allows us to find the number of partial solutions where a forgotten vertex x is coloured 0 by performing some calculations on the number of solutions where x is coloured as a combination of ? and $\hat{0}$.

Lemma 6.5. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$. Let i be a node of $\mathcal{X}$ and $x \in X_i$ some vertex in the bag of i. For any temporal colour c_x, let us use $c \times \{c_x\}$ to indicate a temporal colouring on i where c gives the temporal colours assigned to the vertices $x_{i_j} \in X_i / \{x\}$ and c_x the temporal colour assigned to x. Then:*

$$B_i(c \times \{0\}, \kappa) = B_i(c \times \{?\}, \kappa) / \left(\bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) \right).$$

Proof of Lemma 6.5. We know that $c \times \{0\} \prec c \times \{?\}$ and $c \times \{C\} \prec c \times \{?\}$ for any temporal colour $C \in \{?,\hat{0}\}^\tau / \{?\}$. Therefore, Lemma 6.4 implies that $B_i(c \times \{0\}, \kappa) \subset B_i(c \times \{?\}, \kappa)$ and $\bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) \subset B_i(c \times \{?\}, \kappa)$. We now have to show that $B_i(c \times \{0\}, \kappa)$ and $\bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa)$ form a partition of $B_i(c \times \{?\}, \kappa)$.

First, let us prove that $B_i(c \times \{0\}, \kappa) \cup \bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) = B_i(c \times \{?\}, \kappa)$. Let f be a partial solution of size κ that satisfies $c \times \{?\}$. If for every time step $t \in T$ there is a neighbouring vertex $y \in N_t(x) \cap V_i$ such that $f(y) = 1$ ($R = R_{\text{DOM}}$) or $f(y) = 2$ ($R = R_{\text{ROM}}$), then f satisfies $c \times \{0\}$ and $f \in B_i(c \times \{0\}, \kappa)$. If there is a time step $t \in T$ such that there is no neighbouring vertex $y \in N_t(x) \cap V_i$ such that $f(y) = 1$ ($R = R_{\text{DOM}}$) or $f(y) = 2$ ($R = R_{\text{ROM}}$), then f satisfies $c \times \{c_x\}$ where $c_x^t = \hat{0}$ and $c_x^s = ?$ for every time step $s \neq t$ and $f \in \bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa)$. Therefore $B_i(c \times \{0\}, \kappa) \cup \bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) = B_i(c \times \{?\}, \kappa)$.

Now, we have to prove that $B_i(c \times \{0\}, \kappa) \cap \bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) = \emptyset$. Let $c_x \in \{?,\hat{0}\}^\tau / \{?\}$ be a temporal colour. Let $f \in B_i(c \times \{0\}, \kappa)$. We know that there must be a time step $t \in T$ such that $c_x^t = \hat{0}$. We also know that, since f satisfies $c \times \{0\}$, there is a $y \in N_t(x) \cap V_i$ such that $f(y) = 1$ ($R = R_{\text{DOM}}$) or $f(y) = 2$ ($R = R_{\text{ROM}}$). Therefore f does not satisfy $c \times \{c_x\}$ and $f \notin \bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa)$. Therefore $B_i(c \times \{0\}, \kappa) \cap \bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) = \emptyset$.

Since $B_i(c \times \{0\}, \kappa)$ and $\bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa)$ form a partition of $B_i(c \times \{?\}, \kappa)$, the lemma must hold.

□

Assume that i is a forget node of $\mathcal{X}$ with child j where x is the forgotten vertex and let c be some temporal colouring on i. To find the partial solutions on j that satisfy $c \times \{0\}$, (x is permanently dominated) by taking the solutions where it is coloured ? (possibly dominated) and removing the solutions where it has at least one vertex appearance that is coloured $\hat{0}$ (not dominated). The corresponding table entry is then $A_j(c \times \{0\}, \kappa) = |B_j(c \times \{0\}, \kappa)|$.

However, the algorithm does not have access to B_j, only to A_j. We can find the same result by using an *inclusion exclusion* formula:

Lemma 6.6. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$. Let i be a node of $\mathcal{X}$ and $x \in X_i$ a vertex in the bag of i. For any temporal colour c_x, let us use $c \times \{c_x\}$ to indicate a temporal colouring on i where c gives the temporal colours assigned to the vertices $x_{i_j} \in X_i / \{x\}$ and c_x the temporal colour assigned to x. Then:*

$$A_i(c \times \{0\}, \kappa) = \sum_{C \in \{?,\hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(C)} A_i(c \times \{C\}, \kappa)$$

where $\#_{\hat{0}}(C)$ is the number of vertex appearances $x^t \in V^t$ of x that are given colour $\hat{0}$ by temporal colour C.

Proof of Lemma 6.6. For each time step $t \in T$, let C_t be the temporal colour such that coordinate t has value $\hat{0}$ and all other coordinates have value ?. $C_t^s = \begin{cases} \hat{0} & \text{if } s = t \\ ? & \text{otherwise} \end{cases}$. Now let $B_i^t = B_i(c \times \{C_t\}, \kappa)$. Let f be a partial solution in $\bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa)$, then there is a time step t such that there is no vertex $y \in N_t(x) \cap V_i$ such that $f(y) = 1$ ($R = R_{\text{DOM}}$) or $f(y) = 2$ ($R = R_{\text{ROM}}$). Then $f \in B_i^t$. Therefore $\bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) \subset \bigcup_{t \in T} B_t$. It is now trivial to see that

$$\bigcup_{C \in \{?,\hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) = \bigcup_{t \in T} B_i^t.$$

Also note that, for any positive integer $k \leq \tau$, a partial solution f that is included in $B_i^{t_1} \cap \cdots \cap B_i^{t_k}$ with $t_a \neq t_b$ for each pair of distinct indices $a \neq b$ also satisfies the temporal colouring $c \times \{C\}$ where coordinates $t_1, \cdots, t_k$ of C have colour $\hat{0}$ and all the others have colour ?. Conversely, a partial solution f that satisfies temporal colouring $c \times \{C\}$ where coordinates $t_1, \cdots, t_k$ of C have colour $\hat{0}$ and the others have colour ?, must be included in $B_i^{t_1} \cap \cdots \cap B_i^{t_k}$. Therefore for any positive integer $k \leq \tau$, we have that

$$\sum_{1 \leq t_1 < \cdots < t_k \leq \tau} |B_i^{t_1} \cap \cdots \cap B_i^{t_k}| = \sum_{C \in \{?, \hat{0}\}^\tau, \#_{\hat{0}}(C) = k} |B_i(c \times \{C\}, \kappa)|.$$

Now we will complete the proof by solving the following equation.

$$\begin{aligned}
A_i(c \times \{0\}, \kappa) &= |B_i(c \times \{0\}, \kappa)| \\
&= |B_i(c \times \{?\}, \kappa) / \left(\bigcup_{C \in \{?, \hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa) \right)| \text{ (By Lemma 6.5.)} \\
&= |B_i(c \times \{?\}, \kappa)| - |\bigcup_{C \in \{?, \hat{0}\}^\tau / \{?\}} B_i(c \times \{C\}, \kappa)| \\
&= |B_i(c \times \{?\}, \kappa)| - |\bigcup_{1 \leq t \leq \tau}^{\tau} B_i^t| \\
&= |B_i(c \times \{?\}, \kappa)| - \sum_{1 \leq t \leq \tau} |B_i^t| \text{ (By the inclusion-exclusion principle.)} \\
&\quad + \sum_{1 \leq t < s \leq \tau} |B_i^t \cap B_i^s| \\
&\quad - \sum_{1 \leq t < s < r \leq \tau} |B_i^t \cap B_i^s \cap B_i^r| \\
&\quad + \cdots \\
&\quad + (-1)^\tau |B_i^1 \cap \cdots \cap B_i^\tau| \\
&= \sum_{C \in \{?, \hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(C)} |B_i(c \times \{C\}, \kappa)| \\
&= \sum_{C \in \{?, \hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(C)} A_i(c \times \{C\}, \kappa)
\end{aligned}$$

□

We add the number of partial solutions where the number of vertex appearances that are coloured $\hat{0}$ is even (this includes the number of partial solutions where none of them are) and subtract the number of partial solutions where the number of vertex appearances that are coloured $\hat{0}$ is odd. The inclusion-exclusion formula guarantees that each partial solution where x is coloured ? is counted, but every partial solution where x is coloured $\hat{0}$ in at least one time step is dismissed, while also ensuring we do not dismiss partial solution multiple times.

6.2 Permanent Dominating Set

In this section we will cover the Dominating Set variant of Permanent Domination. The algorithm we present uses the concepts introduced in Section 6.1, as will the algorithm for the Roman Domination variant that we will cover in Section 6.3.

Problem 6.1. Permanent Dominating Set*: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, find the size of a minimum permanent sdf.*

Permanent Dominating Set is equivalent to Simultaneous Domination, sometimes known by the names Global Domination and Factor Domination. These names are not used consistency throughout literature and furthermore the definitions also differ between articles.

The problem known as Factor Domination was first introduced in [4]. In this paper, a *t-factoring* of a graph $H = (V, E)$ into *factors* $G_1, \cdots, G_t$ is defined as a set of graphs $G_i = (V, E_i)$ such that $E_1, \cdots, E_t$ form a partition of edge set E. The problem asks for a minimum size vertex set that is a dominating set on all factors at the same time. Of course, in Permanent Dominating Set, instead of factors, we have the layers of the temporal graph, and the edge sets do not necessarily form a partition. Therefore the original factor domination problem is not equivalent to Permanent Dominating Set.

Later papers have defined the problem in a similar way, but without requiring $E_1 \cdots, E_t$ to be a partition of E. In [6], the problem is defined as Simultaneous Domination and does not require the partition property. Permanent Dominating Set is indeed equivalent to this problem. Since the order of the time steps in Permanent Dominating Set does not influence the solution of the problem we can simply view the layers as factors.

Other papers that discuss these problems include [5], [7], [12], [13] and [19]. As far we know, no algorithm for the problem has been presented before. A task we will take upon ourselves.

We will now use the ground work of the previous subsection to create our first permanent domination algorithm. We use the dynamic programming approach that we desribed before. We visit the nodes of $\mathcal{X}$ from the leaves to the root in a BFS manner. When filling our table, we only use the colours 1, ? and $\hat{0}$. In an sdf, the label 2 is not used, and the colour 0 can be left out, because the only moment where we need to find the number of partial solutions that satisfy a temporal colouring that includes this colour 0, we can calculate the number by using the techniques from Section 6.1.4. There are $2^\tau + 1$ possible temporal colours and thus $(2^\tau + 1)^{n_i}$ possible temporal colourings for each node i.

Any domination function $f : V \to \{0, 1\}$ with $f(v) = 1$ for every vertex v is a valid permanent sdf of size n. Therefore, we can place an upper bound of n on the value of κ.

Algorithm 1: Permanent Dominating Set

PermDom(temporal graph $\mathcal{G}$, nice tree decomposition $\mathcal{X}$ of $G_{\downarrow}$):
  foreach *node i of $\mathcal{X}$ in BFS manner from the leaves to the root r* **do**
    Let A_i be a table with an entry for each colouring c of X_i and value $1 \leq \kappa \leq n$;
    if *i is a leaf node* **then**
      foreach *colouring c of X_i* **do**
        for $\kappa = 1$ *to* n **do**
          if $c = (1)$ *and* $\kappa = 1$ **then**
            $A_i(c, \kappa) = 1$;
          else if $c \neq (1)$ *and* $\kappa = 0$ **then**
            $A_i(c, \kappa) = 1$;
          else
            $A_i(c, \kappa) = 0$;
    else if *i is a forget node* **then**
      Let j be the child of i;
      Let x be the forgotten vertex;
      foreach *colouring c of X_i* **do**
        for $\kappa = 1$ *to* n **do**
          $A_i(c, \kappa) = A_j(c \times \{1\}, \kappa) + \sum_{c_x \in \{?, \hat{0}\}^{\tau}} (-1)^{\#_{\hat{0}}(c_x)} A_j(c \times \{c_x\}, \kappa)$;
    else if *i is an introduce node* **then**
      Let j be the child of i;
      Let x be the introduced vertex;
      foreach *colouring c of X_j* **do**
        for $\kappa = 1$ *to* n **do**
          foreach *temporal colour c_x* **do**
            if $c_x = 1$ **then**
              if $\kappa = 0$ **then**
                $A_i(c \times \{c_x\}, \kappa) = 0$;
              else if *there is a vertex $x_{i_k} \in N_t(x) \cap X_i$ such that $c_k^t = \hat{0}$* **then**
                $A_i(c \times \{c_x\}, \kappa) = 0$;
              else
                $A_i(c \times \{c_x\}, \kappa) = A_j(c, \kappa - 1)$;
            else if *$c_x^t = \hat{0}$ and there is a vertex $x_{i_k} \in N_t(x) \cap X_i$ such that $c_k = 1$* **then**
              $A_i(c \times \{c_x\}, \kappa) = 0$;
            else
              $A_i(c \times \{c_x\}, \kappa) = A_j(c, \kappa)$;
    else
      Let j and k be the children of i;
      foreach *colouring c of X_i* **do**
        for $\kappa = 1$ *to* n **do**
          $A_i(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa + \#_1(c)} A_j(c, \kappa_l) \cdot A_k(c, \kappa_r)$;
  for $\kappa = 1$ *to* n **do**
    if $A(\emptyset, \kappa) > 0$ **then**
      return *κ;*

Each node of $\mathcal{X}$ is either a leaf node, forget node, introduce node or join node. Those nodes will each be handled in the following way:

Leaf Nodes For each leaf node i in $\mathcal{X}$, we have a bag $X_i = (x)$ with only one element. We initialize the table A_i for temporal colouring $c = (c_x)$ and size κ:

$$A_i(c,\kappa) = \begin{cases} 1 & \text{if } c_x = 1 \text{ and } \kappa = 1 \\ 1 & \text{if } c_x \neq 1 \text{ and } \kappa = 0 \\ 0 & \text{otherwise} \end{cases}$$

Lemma 6.7. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a leaf node of $\mathcal{X}$. Then Algorithm 1 correctly calculates the table A_i in $\mathcal{O}(n)$ time.*

Proof of Lemma 6.7. By Definition 2.7, the bag X_i of i has one vertex x. Let $c = (c_x)$ where c_x is the temporal colour for x. Let $f : V_i \to \{0,1\}$ be a partial solution satisfying c. If $c_x = 1$, then we must have that $f(x) = 1$, and x contributes 1 to the value of κ. Moreover, f is the only partial solution that satisfies c. Therefore $A_i((1), 1) = 1$. Since there are no other vertices, the only κ value for which $A_i((1), \kappa) = 1$ is $\kappa = 1$.

If $c_x \neq 1$, then $f(x) = 0$, and x contributes 0 to the value of κ. f is the only partial solution that satisfies c. Therefore $A_i(c, 0) = 1$. Since there are no other vertices, the only κ value for which $A_i(c, \kappa) = 1$ is $\kappa = 0$. Since x does not have neighbours, c cannot be locally invalid.

There are n values for κ that we need to process. For each value of κ, we need to check whether $c_x = 1$ or not, which takes $\mathcal{O}(1)$ time.

Therefore Algorithm 1 correctly calculates A_i in $\mathcal{O}(n)$ time.

□

Forget Nodes Forgotten vertices still impact the tables of nodes yet to come, higher up in $\mathcal{X}$. It is essential that when we forget a vertex x, we build the table of the forget node in a way that only takes into account cases where x was permanently dominated: where x is coloured 1 or 0. Otherwise we cannot guarantee that in the final solution all vertices are dominated.

Let $X_i = \left(x_{i_1}, \cdots, x_{i_{n_i}}\right)$ be the bag of a forget node with child $X_j = \left(x_{i_1}, \cdots, x_{i_{n_i}}, x\right)$, x being the forgotten vertex. Let

$$A_i(c,\kappa) = A_j(c \times \{1\}, \kappa) + \sum_{c_x \in \{?, \hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(c_x)} A_j(c \times \{c_x\}, \kappa)$$

We use the technique of fast subset convolutions from Section 6.1.4 to calculate the number of partial solutions that satisfy a colouring for x where x is coloured 0.

Lemma 6.8. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a forget node of $\mathcal{X}$ with child j. Assume that A_j is correct. Then Algorithm 1 correctly calculates the table A_i in $\mathcal{O}\left(n\left(2^\tau + 1\right)^{n_j}\right)$ time.*

Proof of Lemma 6.8. By Definition 6.2, any domination function $f : V \to \{0,1\}$ such that $f(x) = 0$ and there is a time step such that x has no neighbour with label 1 on that time step, is not a partial solution on i. Since $V_i = V_j$, any partial solution f on j that satisfies $c \times \{0\}$ or $c \times \{1\}$ is also a partial solution on i that satisfies c, since we know for sure that x is permanently dominated by f. Any partial solution on j that satisfies $c \times \{C\}$ such that $\#_{\hat{0}}(C) > 0$ is not a partial solution on i. Therefore we know that

$$\begin{aligned} A_i(c,\kappa) &= A_j(c \times \{1\}, \kappa) + A_j(c \times \{0\}, \kappa) \\ &= A_j(c \times \{1\}, \kappa) + \sum_{c_x \in \{?, \hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(c_x)} A_j(c \times \{c_x\}, \kappa) \text{ (Follows from Lemma 6.6).} \end{aligned}$$

This proves that A_i is calculated correctly if A_j is correct.

As for the complexity, there are $(2^\tau + 1)^{n_i}$ temporal colourings for X_i and n values for κ. For each temporal colouring c, we first take the value of $A_i(c \times \{2\}, \kappa)$ which takes constant time. Finding the value of $\sum_{c_x \in \{?,\hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(c_x)} A_j(c \times \{c_x\}, \kappa)$ requires us to access 2^τ table entries. This brings the total required time to $\mathcal{O}\left(n\,(2^\tau + 1)^{n_i+1}\right) = \mathcal{O}\left(n\,(2^\tau + 1)^{n_j}\right)$.

□

Introduce Nodes When introducing a new vertex, we have to be careful that we do not include locally invalid temporal colourings. Whenever we process a locally invalid temporal colouring we must make sure to make its table entry equal to 0.

Let $X_i = \left(x_{i_1}, \cdots, x_{i_{n_j}}, x\right)$ be the bag of an introduce node that introduces vertex x. Let $X_j = \left(x_{j_1}, \cdots, x_{j_{n_j}}\right)$ be its child. Let $c \times \{c_x\}$ be the temporal colouring for X_i, where c_x is the temporal colour assigned to new vertex x. Let

$$A_i(c \times \{c_x\}, \kappa) = \begin{cases} 0 & \text{if } c_x = 1 \text{ and } \kappa = 0 \\ 0 & \text{else if } c_x = 1 \text{ and there is a vertex } x_{i_k} \in N_t(x) \cap X_i \text{ such that } c_k^t = \hat{0} \\ 0 & \text{else if } c_x^t = \hat{0} \text{ and there is a vertex } x_{i_k} \in N_t(x) \cap X_i \text{ such that } c_k = 1 \\ A_j(c, \kappa - 1) & \text{else if } c_x = 1 \text{ and } \kappa > 0 \\ A_j(c, \kappa) & \text{otherwise} \end{cases}$$

If $c_x = 1$, then vertex x contributes 1 to the value of κ. This means that no partial solution satisfying $c \times \{c_x\}$ is possible where $\kappa = 0$. If $\kappa > 0$, we have to look in the table of j in the entries for c and $\kappa - 1$ since x cannot contribute to the size yet in node j. We also need to check whether x has any neighbouring vertex appearances that are coloured $\hat{0}$. If it does, this colouring is locally invalid. In the same vein, when x itself has an appearance that is coloured $\hat{0}$, we should check if that vertex appearance has a neighbour that is coloured 1, in which case this is also locally invalid. Otherwise, if $c_x \neq 1$, then we simply take the table entry for c and κ on node j. Any partial solution on j of size κ that satisfies c can be transformed into a partial solution on i of the same size that satisfies $c \times \{c_x\}$ by adding x to the domain and setting $f(x) = 0$. in fact, these partial solutions are the only partial solutions on i that satisfy this temporal colouring.

Lemma 6.9. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a introduce node of $\mathcal{X}$ with child j. Assume that A_j is correct. Then Algorithm 1 correctly calculates the table A_i in $\mathcal{O}\left(n\,(2^\tau + 1)^{n_i}\right)$ time.*

Proof of Lemma 6.9. Whenever $c \times \{c_x\}$ is locally invalid, no partial solution can satisfy it. Therefore $A_i(c \times \{c_x\}, \kappa)$ should be 0 for any κ. It is easy to see that the formula used by Algorithm 1 satisfies this condition. We know by Definition 2.7 that x cannot have neighbours outside X_i, therefore if $c \times \{c_x\}$ is not locally invalid, any partial solution f on j that satisfies c can be transformed into a partial solution f' on i that satisfies $c \times \{c_x\}$ if we make f' identical to f on V_j and add x to the domain and set $f'(x) = 1$ if $c_x = 1$ and $f'(x) = 0$ if $c_x \in \left\{?, \hat{0}\right\}^\tau$. A domination function that was not constructed in this way, does not satisfy $c \times \{c_x\}$ and thus need not be taken into account. In the case where $c_x = 1$, such a partial solution f on j must be of size $\kappa - 1$, since adding x to the domain with $f(x)$ would increase the size of f by 1, making its size $|f| + 1 = \kappa - 1 + 1 = \kappa$. This proves that Algorithm 1 calculates A_i correctly if A_j is correct.

As for the complexity, for each temporal colouring and value of κ, we have to perform a simple check and we possibly access a single table entry. Therefore, the required time is $\mathcal{O}\left(n\,(2^\tau + 1)^{n_i}\right)$.

□

Join Nodes Let $X_i = \left(x_{i_1}, \cdots, x_{i_{n_i}}\right)$ be the bag of a join node with children j, k with $X_i = X_j = X_k$. Let

$$A_i(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa + \#_1(c)} A_j\left(c, \kappa_l\right) \cdot A_k\left(c, \kappa_r\right)$$

where $\#_1(c) := |\{c_a \in c : c_a = 1\}|$. Each child of the join node has the same bag, but their history vertex sets can differ. If we want our partial solution to be of size κ, then we have a partial solution for each combination of a partial solution on j of size κ_l and a partial solution on k of size κ_r all satisfying colouring c, such that $\kappa_l + \kappa_r = \kappa + \#_1(c)$. We add $\#_1$ to the right side of the equation because each vertex that is coloured 1 adds 1 to the value of κ, and otherwise we would count the vertices from X_i that are coloured 1 twice.

Lemma 6.10. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a join node of $\mathcal{X}$ with children j and k. Assume that A_j and A_k are correct. Then Algorithm 1 correctly calculates the table A_i in $\mathcal{O}\left(n^2\left(2^\tau + 1\right)^{n_i}\right)$ time.*

Proof of Lemma 6.10. Let c be a temporal colouring on X_i and let f_l and f_r be partial solutions on j and k respectively that both satisfy c. Then let the domination function $f_l \oplus f_r : V_i \to \{0, 1\}$ be defined as:

$$f_l \oplus f_r(x) = \begin{cases} f_l(x) & \text{if } x \in V_j \\ f_r(x) & \text{otherwise.} \end{cases}$$

Since f_l and f_r both satisfy c, they both assign the same value to each vertex in X_i. Definition 2.6 implies that $V_j \cap V_k = X_i$. For any vertex $x \in V_j/X_i$, $f_l \oplus f_r$ assigns the value given to x by f_l and for any vertex $x \in V_k/X_i$, $f_l \oplus f_r$ assigns the value given to x by f_r. We know that every vertex $x \in V_i/X_i$ is dominated by $f_l \oplus f_r$ because they are dominated in their respective history vertex set by the partial solution associated with that node. Therefore we know that $f_l \oplus f_r$ is a partial solution on i that satisfies c. The size of $f_l \oplus f_r$ is $|f_l \oplus f_r| = |f_l| + |f_r| - \#_1(c)$. We subtract $\#_1(c)$ since each label for vertices outside X_i is counted once in $|f_l| + |f_r|$ and each label for vertices in X_i is counted twice. If $|f_l \oplus f_r| = \kappa$, then $f_l \oplus f_r \in B_i(c, \kappa)$.

For any temporal colouring c of X_i and non-negative integers κ_l and κ_r, let $B_j(c, \kappa_l) \oplus B_k(c, \kappa_r) = \{f_l \oplus f_r : f_l \in B_j(c, \kappa_l), f_r \in B_k(c, \kappa_r)\}$ (let $B_j(c, \kappa_l) \oplus B_k(c, \kappa_r) = \emptyset$ if one of the two sets is empty). Then we know that for any κ_l and κ_r such that $\kappa_l + \kappa_r = \kappa + 2\#_2(c)$, we must have that $B_j(c, \kappa_l) \oplus B_k(c, \kappa_r) \subset B_i(c, \kappa)$.

Now let $f : V_i \to \{0, 1\}$ be a partial solution on i that satisfies c. Let $f_l : V_j \to \{0, 1\}$ be f limited to the domain V_j and let $f_r : V_k \to \{0, 1\}$ be f limited to the domain V_k. Then f_l and f_r are partial solutions on j and k respectively that satisfy c and $f = f_l \oplus f_r$. Therefore

$$B_i(c, \kappa) \subset \bigcup_{\kappa_l + \kappa_r = \kappa + \#_1(c)} B_j(c, \kappa_l) \oplus B_k(c, \kappa_r)$$

and since we have also proven that $B_j(c, \kappa_l) \oplus B_k(c, \kappa_r) \subset B_i(c, \kappa)$ for each $B_j(c, \kappa_l) \oplus B_k(c, \kappa_r)$ with $\kappa_l + \kappa_r = \kappa + \#_1(c)$, we now have that

$$B_i(c, \kappa) = \bigcup_{\kappa_l + \kappa_r = \kappa + \#_1(c)} B_j(c, \kappa_l) \oplus B_k(c, \kappa_r).$$

Next, note that $B_j(c, \kappa_l) \oplus B_k(c, \kappa_r)$ has a unique element for each pair of unique elements f_l and f_r of $B_j(c, \kappa_l)$ and $B_k(c, \kappa_r)$ respectively. Therefore $|B_j(c, \kappa_l) \oplus B_k(c, \kappa_r)| = A_j(c, \kappa_l) \cdot A_k(c, \kappa_r)$. Finally, note that $B_j(c, \kappa_{l_1}) \oplus B_k(c, \kappa_{r_1}) \cap B_j(c, \kappa_{l_2}) \oplus B_k(c, \kappa_{r_2}) = \emptyset$ if $\kappa_{l_1} \neq \kappa_{l_2}$ and $\kappa_{r_1} \neq \kappa_{r_2}$. This allows us to prove the lemma with the following equation.

$$\begin{aligned}
A_i(c,\kappa) &= |B_i(c,\kappa)| \\
&= |\bigcup_{\kappa_l+\kappa_r=\kappa+\#_1(c)} B_j(c,\kappa_l) \oplus B_k(c,\kappa_r)| \\
&= \sum_{\kappa_l+\kappa_r=\kappa+\#_1(c)} |B_j(c,\kappa_l) \oplus B_k(c,\kappa_r)| \\
&= \sum_{\kappa_l+\kappa_r=\kappa+\#_1(c)} A_j(c,\kappa_l) \cdot A_k(c,\kappa_r)
\end{aligned}$$

We have now proven that Algorithm 1 calculates A_i correctly if A_j and A_k are correct.

As for the complexity, for each value of c and κ, we perform $\mathcal{O}(n)$ multiplications. Therefore this step can be performed in $\mathcal{O}\left(n^2 (2^\tau + 1)^{n_i}\right)$ time.

□

Termination Let r be the root node of $\mathcal{X}$. Its bag $X_r = \emptyset$ is empty. Once the table of r has been filled, we have explored the entire tree decomposition. We now return the minimum κ for which $A_r(\emptyset, \kappa) > 0$. This can be done in $\mathcal{O}(n)$ time.

Theorem 6.1. *Algorithm 1 solves* Permanent Dominating Set *correctly.*

Proof of Theorem 6.1. Let L_1 be the set of leaf nodes of $\mathcal{X}$. For any integer $l > 1$, let L_l be the set of nodes of $\mathcal{X}$ such that that have at least one child in L_{l-1}. We will refer to nodes in L_l to *level i nodes*. Note that a join node can be part of multiple levels. For simplicity, let a join node i with children in L_a and L_b only appear in level $\max(a, b) + 1$.

We will prove that the table A_i of any node i of $\mathcal{X}$ is correct for the considered colourings (igoring the colours 1 and 0). We will prove this by induction on the level l. If $l = 1$, then Lemma 6.7 shows that the tables of all nodes in L_l are correct. Now we show that if the tables of all nodes in $L_1 \cup \cdots \cup L_{k-1}$ are correct, then the tables of all nodes in L_k are correct. Let i be a node in L_k. i must either be a forget node, an introduce node or a join node. Lemmas 6.8, 6.9 and 6.10 show that in all three cases, A_i must be correct.

This means that the table A_r for the root node r is also correct. Algorithm 1 returns the κ for which $A_r(\emptyset, \kappa) > 0$. By Definition 2.7 we have that $V_r = V$ and furthermore we have $X_r = \emptyset$. Therefore, by Definition 6.2, any partial solution f on r must permanently dominate every vertex in V_r and is, thus, a permanent sdf on $\mathcal{G}$. Therefore Algorithm 1 correctly solves Permanent Dominating Set.

□

Now that we have proven the correctness of Algorithm 1, the only thing left to prove is the complexity of the algorithm.

Theorem 6.2. Permanent Dominating Set *can be solved in* $\mathcal{O}\left(n^3 (2^\tau + 1)^{\ell+1}\right)$ *time and* $\mathcal{O}\left(n (2^\tau + 1)^{\ell+1}\right)$ *space, where* ℓ *is the treewidth of the underlying graph.*

Proof of Theorem 6.2. The join nodes are the bottleneck of this algorithm. There are at most $\mathcal{O}(n)$ nodes in $\mathcal{X}$ [24]. The total runtime of this algorithm therefore is $\mathcal{O}\left(n^3 (2^\tau + 1)^{\ell+1}\right)$.

For each node i, there can be $(2^\tau + 1)^{n_i}$ temporal colourings and n values for κ. Since there are at moast $\mathcal{O}(n)$ nodes in $\mathcal{X}$, the algorithm requires $\mathcal{O}\left(n (2^\tau + 1)^{\ell+1}\right)$ space.

□

6.3 Permanent Roman Domination

This section will cover the Roman Domination variant of Permanent Domination. The algorithm we present uses the same concepts from Section 6.1 that we used for the algorithm in Section 6.2.

Problem 6.2. Permanent Roman Domination: *Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$, find the size of a minimum permanent rdf.*

The algorithm for Permanent Roman Domination is very similar to the one for Permanent Dominating Set. The main difference is that vertices can now be assigned label 1 or 2 by a partial solution. The label 2 takes the role previously filled by label 1, while label 1 now signifies a vertex that dominates only itself. By using techniques from [30] and [32], we can ignore the label 1 for the most part. Only when we forget a vertex, we will take into account the partial solutions that label this vertex 1. If the forgotten vertex x is assigned any temporal colour $C \in \left\{?, \hat{0}\right\}^{\tau}$, we could replace that temporal colour by 1, ensuring that x is permanently dominated.

When introducing a vertex appearance $v^t \in V^t$ that is coloured $C \in \left\{?, \hat{0}\right\}^{\tau}$, we pretend as if $f(v) = 0$ and as if it does not contribute anything to κ. However, when forgetting a vertex v that is not coloured 2, we do take into account the possibility that $f(v) = 1$.

Since assigning label 1 to each vertex $v \in V$ is enough to permanently dominate $\mathcal{G}$, we can put an upper bound on the value of κ again.

Algorithm 2: Permanent Roman Domination

`PermRom(temporal graph` $\mathcal{G}$`, nice tree decomposition` $\mathcal{X}$ `of` $G_\downarrow$`):`
- **foreach** *node i of $\mathcal{X}$ in BFS manner from the leaves to the root r* **do**
  - Let A_i be a table with an entry for each colouring c of X_i and value $1 \leq \kappa \leq n$;
  - **if** *i is a leaf node* **then**
    - **foreach** *colouring c of X_i* **do**
      - **for** $\kappa = 1$ *to* n **do**
        - **if** $c = (2)$ *and* $\kappa = 2$ **then**
          - $A_i(c, \kappa) = 1$;
        - **else if** $c \neq (2)$ *and* $\kappa = 0$ **then**
          - $A_i(c, \kappa) = 1$
        - **else**
          - $A_i(c, \kappa) = 0$;
  - **else if** *i is a forget node* **then**
    - Let j be the child of i;
    - Let x be the forgotten vertex;
    - **foreach** *colouring c of X_i* **do**
      - **for** $\kappa = 1$ *to* n **do**
        - $A_i(c, \kappa) = A_j(c \times \{2\}, \kappa) + \sum_{c_x \in \{?, \hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(c_x)} A_j(c \times \{c_x\}, \kappa) + A_j(c \times \{?\}, \kappa - 1)$;
  - **else if** *i is an introduce node* **then**
    - Let j be the child of i;
    - Let x be the introduced vertex;
    - **foreach** *colouring c of X_j* **do**
      - **for** $\kappa = 1$ *to* n **do**
        - **foreach** *temporal colour c_x* **do**
          - **if** $c_x = 2$ **then**
            - **if** $\kappa < 2$ **then**
              - $A_i(c \times \{c_x\}, \kappa) = 0$;
            - **else if** *there is a vertex $x_{i_k} \in N_t(x) \cap X_i$ such that $c_k^t = \hat{0}$* **then**
              - $A_i(c \times \{c_x\}, \kappa) = 0$;
            - **else**
              - $A_i(c \times \{c_x\}, \kappa) = A_j(c, \kappa - 2)$;
          - **else if** $c_x^t = \hat{0}$ *and there is a vertex $x_{i_k} \in N_t(x) \cap X_i$ such that $c_k = 2$* **then**
            - $A_i(c \times \{c_x\}, \kappa) = 0$;
          - **else**
            - $A_i(c \times \{c_x\}, \kappa) = A_j(c, \kappa)$;
  - **else**
    - Let j and k be the children of i;
    - **foreach** *colouring c of X_i* **do**
      - **for** $\kappa = 1$ *to* n **do**
        - $A_i(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa + 2\#_2(c)} A_j(c, \kappa_l) \cdot A_k(c, \kappa_r)$;
- **for** $\kappa = 1$ *to* n **do**
  - **if** $A(\emptyset, \kappa) > 0$ **then**
    - **return** *κ;*

Leaf Nodes For each leaf node i in $\mathcal{X}$, we have a bag $X_i = (x)$ with only one element. Since we ignore the label 1 for now, we initialize the table A_i for temporal colouring $c = (c_x)$ and size κ:

$$A_i(c,\kappa) = \begin{cases} 1 & \text{if } c_x = 2 \text{ and } \kappa = 2 \\ 1 & \text{if } c_x \neq 2 \text{ and } \kappa = 0 \\ 0 & \text{otherwise} \end{cases}$$

Lemma 6.11. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a leaf node of $\mathcal{X}$. Then Algorithm 2 correctly calculates the table A_i in $\mathcal{O}(n)$ time.*

Proof of Lemma 6.11. We do not use the label 1 in leaf nodes in this problem. The label 2 takes the role that label 1 took in the PERMANENT DOMINATING SET problem. Therefore it is trivial to see that the proof for Lemma 6.7 also works for this lemma, if we replace the mentions of the label 1 by the label 2, and consider that if $f(x) = 2$, then x contributes 2 to the value of κ.

□

Forget Nodes Let $X_i = \left(x_{i_1}, \cdots, x_{i_{n_i}}\right)$ be the bag of a forget node with child $X_j = \left(x_{i_1}, \cdots, x_{i_{n_i}}, x\right)$, x being the forgotten vertex. Let

$$A_i(c,\kappa) = A_j(c \times \{2\}, \kappa) + \sum_{c_x \in \{?, \hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(c_x)} A_j(c \times \{c_x\}, \kappa) + A_j(c \times \{?\}, \kappa - 1)$$

The first addent takes into account the temporal colourings where x is coloured 2, the second addent takes into account the temporal colourings where x is coloured 0, and the final addent takes into account the number of partial solutions f on j where x gets label $f(x) = 0$ of size $\kappa - 1$, which is equivalent to the temporal colourings where x is coloured 1. These represent the partial solutions f' on i where $f'(x) = 1$ and $f'(v) = f(v)$ for all vertices $v \neq x$. We look up table entries of size $\kappa - 1$ instead of κ, because x *should* contribute 1 to the size of the partial solution if it is going to receive label 1, but does not do so yet in A_j, since it is regarded as a vertex with label 0.

Lemma 6.12. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a forget node of $\mathcal{X}$ with child j. Assume that A_j is correct. Then Algorithm 2 correctly calculates the table A_i in $\mathcal{O}\left(n\left(2^\tau + 1\right)^{n_j}\right)$ time.*

Proof of Lemma 6.12. Let i be a forget node with child j, and let x be the vertex that is forgotten. Let c be a temporal colouring on X_i and let $0 \leq \kappa \leq n$. Replacing the label 1 by the label 2, Lemma 6.8 shows that any partial solution of size κ on j that satisfies $c \times \{2\}$ or $c \times \{0\}$, is also a partial solution on i. Furthermore, any partial solution of size κ on j that satisfies $c \times \{1\}$ is also a partial solution on i, since then we also know that x is permanently dominated (by itself). However, the algorithm does not have access to $A_j(c \times \{1\}, \kappa)$ since we do not use the colour 1. There is another way to calculate this value.

We introduce the transformation $\lambda : B_j(c \times \{?\}, \kappa - 1) \to B_j(c \times \{1\}, \kappa)$. For any partial solution f on j of size $\kappa - 1$ that satisfies $c \times \{?\}$, let $\lambda f : V_i \to \{0, 1, 2\}$ be a domination function defined as:

$$\lambda f(v) = \begin{cases} 1 & \text{if } v = x \\ f(v) & \text{otherwise.} \end{cases}$$

We know that all vertices $v \in X_i$ other than x receive the label $\lambda f(v) = f(v)$ assigned to them by c. Any vertex $v \in V_j / X_j$ is permanently dominated by λf since it is also permanently dominated by f, and x does not dominate any other vertex in either f or λf. Since x now contributes 1 to the value of $|\lambda f|$, increasing it by 1 compared to $|f|$, we have that $|\lambda f| = \kappa$. Therefore, if $f \in B_j(c \times \{?\}, \kappa - 1)$ then $\lambda f \in B_j(c \times \{1\}, \kappa)$. Also, it is clear that if for two partial solutions $f, g \in B_j(c \times \{?\}, \kappa - 1)$ we have $f \neq g$, then also $\lambda f \neq \lambda g$ since they must differ on a vertex other than x and thus $A_j(c \times \{?\}, \kappa - 1) \leq A_j(c \times \{1\}, \kappa)$.

It is easy to see that this same argument on the inverse transformation $\lambda^{-1} : B_j(c\times\{1\}, \kappa) \to B_j(c\times\{?\}, \kappa-1)$ implies that $A_j(c\times\{?\}, \kappa-1) \geq A_j(c\times\{1\}, \kappa)$ and thus $A_j(c\times\{?\}, \kappa-1) = A_j(c\times\{1\}, \kappa)$. Therefore we now have:

$$\begin{aligned} A_i(c, \kappa) &= A_j(c \times \{2\}, \kappa) + A_j(c \times \{1\}, \kappa) + A_j(c \times \{0\}, \kappa) \\ &= A_j(c \times \{2\}, \kappa) + A_j(c \times \{?\}, \kappa - 1) + \sum_{c_x \in \{?, \hat{0}\}^\tau} (-1)^{\#_{\hat{0}}(c_x)} A_j(c \times \{c_x\}, \kappa) \end{aligned}$$

This means that A_i is correctly calculated by Algorithm 2 if A_j is correct.

Compared to Algorithm 1, we only consider one extra table entry of A_j per table entry of A_i in a forget node. Therefore the time complexity remains the same as for forget nodes in that algorithm.

□

Introduce Nodes Let $X_i = \left(x_{i_1}, \cdots, x_{i_{n_j}}, x\right)$ be the bag of an introduce node that introduces vertex x. Let $X_j = \left(x_{j_1}, \cdots, x_{j_{n_j}}\right)$ be its child. Again, we ignore the label 1. Let $c \times \{c_x\}$ be the temporal colouring for X_i, where c_x is the temporal colour assigned to new vertex x. Let

$$A_i(c\times\{c_x\}, \kappa) = \begin{cases} 0 & \text{if } c_x = 2 \text{ and } \kappa < 2 \\ 0 & \text{else if } c_x = 2 \text{ and there is a vertex } x_{i_k} \in N_t(x) \cap X_i \text{ such that } c_k^t = \hat{0} \\ 0 & \text{else if } c_x^t = \hat{0} \text{ and there is a vertex } x_{i_k} \in N_t(x) \cap X_i \text{ such that } c_k = 2 \\ A_j(c, \kappa - 2) & \text{else if } c_x = 2 \text{ and } \kappa > 0 \\ A_j(c, \kappa) & \text{otherwise} \end{cases}$$

Lemma 6.13. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a introduce node of $\mathcal{X}$ with child j. Assume that A_j is correct. Then Algorithm 2 correctly calculates the table A_i in $\mathcal{O}\left(n\left(2^\tau + 1\right)^{n_i}\right)$ time.*

Proof of Lemma 6.13. The proof of Lemma 6.9 also works here, if we replace mentions of label 1 with label 2, and consider that any vertex x with $f(x) = 2$ contributes 2 to the size of κ.

□

Join Nodes Let $X_i = \left(x_{i_1}, \cdots, x_{i_{n_i}}\right)$ be the bag of a join node with children j, k with $X_i = X_j = X_k$. Let

$$A_i(c, \kappa) = \sum_{\kappa_l + \kappa_r = \kappa + 2\#_2(c)} A_j\left(c, \kappa_l\right) \cdot A_k\left(c, \kappa_r\right)$$

where $\#_2(c) := |\{c_a \in c : c_a = 2\}|$. We add $2\#_2$ to the right side of the equation because each vertex that is coloured 2 adds 2 to the value of κ, and otherwise we would count the vertices from X_i that are coloured 2 twice.

Lemma 6.14. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$ and i a join node of $\mathcal{X}$ with children j and k. Assume that A_j and A_k are correct. Then Algorithm 2 correctly calculates the table A_i in $\mathcal{O}\left(n^2\left(2^\tau + 1\right)^{n_i}\right)$ time.*

Proof of Lemma 6.14. The proof of Lemma 6.10 also works here, if we replace mentions of label 1 with label 2, and consider that any vertex x with $f(x) = 2$ contributes 2 to the size of κ.

□

Termination This happens in the same way as in Algorithm 1.

Theorem 6.3. *Algorithm 2 solves* PERMANENT ROMAN DOMINATION *correctly.*

Proof of Theorem 6.3. The proof of Theorem 6.1 in combination with the proofs for Lemmas 6.11, 6.12, 6.13 and 6.14 imply that the partial solutions on the root node are permanent rdf's. Therefore Algorithm 2 correctly solves PERMANENT ROMAN DOMINATION.

□

Theorem 6.4. PERMANENT ROMAN DOMINATION *can be solved in* $\mathcal{O}\left(n^3\left(2^\tau+1\right)^{\ell+1}\right)$ *time and* $\mathcal{O}\left(n\left(2^\tau+1\right)^{\ell+1}\right)$ *space, where* ℓ *is the treewidth of the temporal graph.*

Proof of Theorem 6.4. All nodes are handled with the same time and space complexity as in Algorithm 1.

□

7 PERIODIC DOMINATION

This section covers the PERIODIC DOMINATION requirement, applied to DOMINATING SET and ROMAN DOMINATION. The problems are as follows:

Problem 7.1. PERIODIC DOMINATING SET*:* *Given a temporal graph* $\mathcal{G} = (V, E, \tau, \lambda)$*, find the size of a minimum periodic sdf.*

Problem 7.2. PERIODIC ROMAN DOMINATION*:* *Given a temporal graph* $\mathcal{G} = (V, E, \tau, \lambda)$*, find the size of a minimum periodic rdf.*

Though the problems that we have analyzed so far are indeed problems on temporal graphs, one might argue that they are not "temporal" in the true definition of the word. We have defined these problems using the set of layers and their appropriate edge sets, but we do not need all the information in temporal graphs to do so. We could have simply used a set of *factors* instead of layers: graphs sharing the same vertex set that appear in no particular order. The order of the time steps has no consequence of the solution of any of those problems. PERMANENT DOMINATING SET has already been studied previously in this way under the name SIMULTANEOUS DOMINATION. If the order of the layers is of no consequence, are we truly dealing with a temporal problem? The word "temporal", after all, relates to time specifically. Not just the events but also the order in which they occur are essential when talking about time.

The reason why the order of the time steps is important for the solution to PERIODIC DOMINATION, is that the order of the time steps defines the time windows defined in Definition 4.4. The order of time windows themselves, however is not important. This allows us to solve these problem by transforming $\mathcal{G}$ into a new temporal graph.

Definition 7.1. *Let* $\mathcal{G} = (V, E, \tau, \lambda)$ *be a temporal graph,* θ *a period and* $G_1^\theta, \cdots, G_{\tau+1-\theta}^\theta$ *the windows of* $\mathcal{G}$*. Then the temporal graph* $\mathcal{G}^\theta = \left(G_1^\theta, \cdots, G_{\tau+1-\theta}^\theta\right)$ *is called the* ***θ-collapsed time window graph****, or simply the* ***θ-collapsed graph*** *of* $\mathcal{G}$*.*

$\mathcal{G}^\theta$ is a temporal graph where each layer is one of the windows. We can solve PERMANENT DOMINATING SET, PERMANENT ROMAN DOMINATION on $\mathcal{G}^\theta$ to get solutions to PERIODIC DOMINATING SET and PERIODIC ROMAN DOMINATION respectively.

Lemma 7.1. *Let* $\mathcal{G} = (V, E, \tau, \lambda)$ *be a temporal graph,* R *a domination rule,* θ *a period and* $G_1^\theta, \cdots, G_{\tau+1-\theta}^\theta$ *the windows of* $\mathcal{G}$*. Then solving* PERIODIC DOMINATION *for* $\mathcal{G}$ *and* R *is equivalent to solving* PERMANENT DOMINATION *for* $\mathcal{G}^\theta$ *and* R*.*

Proof of Lemma 7.1. It is easy to see from Definitions 4.3 and 4.5 that any R-periodic function for $\mathcal{G}$ is also an R-permanent function for $\mathcal{G}^\theta$ and vice versa. Therefore any minimum R-permanent function on $\mathcal{G}^\theta$ is also a minimum R-periodic function for $\mathcal{G}$.

□

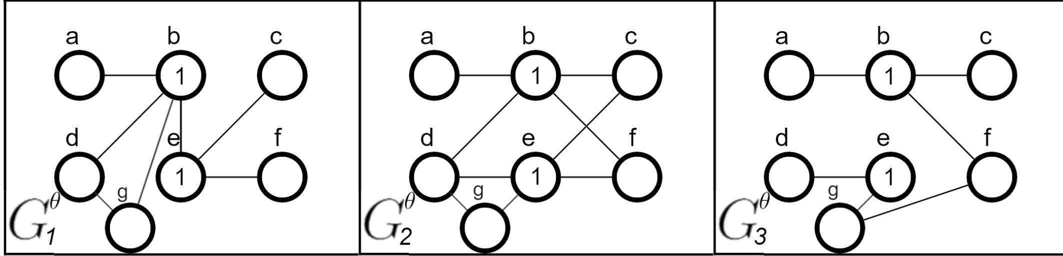

Figure 7.1: The collapsed graph corresponding to the temporal graph from Figure 4.1 and $\theta = 2$. The solution to R_{Dom}-Permanent Domination on the collapsed graph also solves R_{Dom}-Periodic Domination on the original graph.

Theorem 7.1. Periodic Dominating Set *and* Periodic Roman Domination *can be solved in* $\mathcal{O}\left(n^3\left(2^{\tau+1-\theta}+1\right)^{\ell+1}\right)$ *time and* $\mathcal{O}\left(n\left(2^{\tau+1-\theta}+1\right)^{\ell+1}\right)$ *space, where* ℓ *is the treewidth of the underlying graph* $G_\downarrow$.

Proof of Theorem 7.1. Transforming $\mathcal{G}$ into $\mathcal{G}^\theta$ takes $\mathcal{O}(\tau\theta)$ time, since we need to gather θ edge sets and unify them $\tau+1-\theta$ times. The lifetime of $\mathcal{G}^\theta$ is $\tau+1-\theta$. Furthermore, $\mathcal{G}^\theta_\downarrow = G_\downarrow$ because they have the same vertex and edge set. Let $\mathcal{X}$ be a nice tree decomposition of $G_\downarrow$ of width ℓ. Then solving Permanent Dominating Set or Permanent Roman Domination on $\mathcal{G}^\theta$ takes $\mathcal{O}\left(n^3\left(2^{\tau+1-\theta}+1\right)^{\ell}\right)$ time by Theorems 6.2 and 6.4. By Lemma 7.1, the solutions to these problems are the same as for Periodic Dominating Set and Periodic Roman Domination on $\mathcal{G}$.

□

8 Evolving Domination

This section covers Evolving Domination. For specific domination rules, the problems are as follows:

Problem 8.1. Evolving Dominating Set: *Given a temporal graph* $\mathcal{G} = (V, E, \tau, \lambda)$*, find the total size* $\sum_{t\in T}|f_t|$ *of a sequence of domination functions* $(f_1, \cdots, f_\tau)$ *such that* f_t *is a minimum sdf for* G_t.

Problem 8.2. Evolving Roman Domination: *Given a temporal graph* $\mathcal{G} = (V, E, \tau, \lambda)$*, find the total size* $\sum_{t\in T}|f_t|$ *of a sequence of domination functions* $(f_1, \cdots, f_\tau)$ *such that* f_t *is a minimum rdf for* G_t.

Problem 8.3. Evolving Weak Roman Domination: *Given a temporal graph* $\mathcal{G} = (V, E, \tau, \lambda)$*, find the total size* $\sum_{t\in T}|f_t|$ *of a sequence of domination functions* $(f_1, \cdots, f_\tau)$ *such that* f_t *is a minimum wdf for* G_t.

8.1 Maintenance Algorithms

Up until now all problems we have covered ask for a single domination function that dominates all vertices in some way. Evolving Domination is quite different. It asks for a separate domination function for each time step. In essence it asks us to solve τ static problems. There is an obvious way to solve this problem.

Theorem 8.1. *Let h be some computable function and $R \in \{R_{\text{Dom}}, R_{\text{Rom}}, R_{\text{Weak}}\}$ a domination rule. Let A_R be an algorithm that solves R-*Domination *and let T_R be a function such that the required running time for A_R for input X is $\mathcal{O}(T_R(X))$. Then Evolving Domination for R can be solved in $\mathcal{O}(\tau T_R(X))$ time where X is the input corresponding to the layer that takes the largest amount of time for A_R.*

The proof for this theorem is trivial. We simply apply A_R to each layer to get the size of a minimum domination function. Then we return the sum of thoses sizes. This method of solving Evolving Domination will be referred to as the *trivial solution*. The main question regarding Evolving Domination is if it is possible to do it faster. Casteigts and Flocchini [8] proposed a question regarding Evolving Domination. After rephrasing for generality, this question becomes:

"Given any temporal graph $\mathcal{G}$ and a minimum R-domination function for layer G_i, is it possible to compute a minimum R-domination function for layer G_{i+1} more quickly than computing it from scratch?"

By "computing it from scratch", we specifically mean computing it with the current fastest algorithm A'_R that does so with running time $\mathcal{O}(T'_R(X))$ for input X. If we can beat the running time of A'_R by using the solution to the previous layer, we can improve the required time of $\mathcal{O}(\tau T'_R(X))$ required for Evolving Domination. There is no guarantee, however, that the layers of a temporal graph are similar in any way, except for the fact that they all have the same number of vertices. Therefore this question is equivalent to:

"Given any two static graphs G and H with the same number of vertices and a minimum R-domination function for H, is it possible to compute a minimum R-domination function for G more quickly than computing it from scratch?"

The answer to this last question is obviously "No." It may be true in some cases, but if it were true for *any* two static graphs of the same size then, in particular, for any algorithm A_R for R-Domination that runs in $\mathcal{O}(T_R(X))$ time, we can solve R-Domination on any graph $G = (V, E)$ faster than in $\mathcal{O}(T_R(X))$ time by taking a minimum R-domination function (trivial) for $H = (V, \emptyset)$ and applying some black box algorithm to it. The obtained running time could then, again, be improved in the same way, and so on.

Casteigts and Flocchini then specificially pose the question for temporal graphs where the layers are *neighbour graphs*.

Definition 8.1. *Let $G = (V, E)$ be a static graph and $e \in E$ some edge. Then G and $H = (V, E/\{e\})$ are each other's* ***neighbour graphs****. In addition, any static graph is its own neighbour graph.*

An algorithm that solves a problem on some static graph G by using a known solution to the same problem on a neighbour graph, is called a *maintenance algorithm*.

Definition 8.2. *Let R be a domination rule. A* ***maintenance algorithm*** *is an algorithm that computes a minimum R-domination function for any static graph G given a minimum R-domination function on any neighbour graph H of G.*

To use a maintenance algorithm on a temporal graph, we require all the layers to be neighbour graphs of their predecessor and successor. Such a graph will be called a *cozy temporal graph*.

Definition 8.3. *Let $\mathcal{G} = (G_1, \cdots, G_\tau)$ be a temporal graph. If G_t and G_{t+1} are neighbour graphs for each $t < \tau$, then we call $\mathcal{G}$ a* ***cozy temporal graph***[2].

Once again, let A'_R be the current fastest *constructive* algorithm that solves R-domination, and let $\mathcal{O}(T'_R(X))$ be its running time for input X. Let B_R be a maintenance algorithm that finds a minimum R-domination function on a static graph G given a minimum R-domination function on a neighbour graph H in $\mathcal{O}(S_R(X))$ time for input X and assume that this is faster than $\mathcal{O}(T'_R(X))$. On a cozy temporal graph $\mathcal{G} = (G_1, \cdots, G_\tau)$ we could then use A'_R to find a minimum R-domination function for G_1 and use B_R for every other layer. The required running time would then be $\mathcal{O}(T'_R(X_1) + \tau S_R(X))$ time where X_1 is the input corresponding to layer G_1 and X is the input corresponding to the layer that takes the largest amount of time for B_R. X includes the corresponding layer graph and a minimum R-domination function for the previous layer.

[2] Hjuler et al. [22] use the term *dynamic graph* instead of *cozy graph*, though other articles use that term to refer to temporal graphs in general, or to a graph structure not only the edges but also the vertices can appear and disappear each time step. To avoid confusion, we have decided to use a new name.

Note that our definition of a maintenance algorithm involves finding an actual minimum R-domination function, not just the *size* of one. In other words, it has to be *constructive*. In the above example we also require A'_R to be constructive. We include these requirement because in order to use a maintenance algorithm, we need a solution to the previous layer.

As Casteigts et al. [9] show, any $\mathcal{NP}$-hard problem Π cannot have a polynomial maintenance algorithm unless $P = \mathcal{NP}$. Indeed, if Π did have such an algorithm then we could find a solution on any static graph $G = (V, E)$ by first finding trivial solution S_0 on $G_0 = (V, \emptyset)$ and add the edges in E step by step to obtain $G_i = (V, E_i)$ where $E = \{e_1, \cdots, e_m\}$ and $E_i = \{e_1, \cdots, e_i\}$ and maintain the solution S_i for every step using this maintenance algorithm until we find solution S_m for $G_m = G$. This would take polynomial time. Furthermore, they show that any problem Π that is $W[1]-hard$ for parameter k cannot have an $\mathcal{O}(f(k)p(n))$ time maintenance algorithm where f is any computable function and p is a polynomial, unless $W[1] = FPT$. Therefore, to improve EVOLVING DOMINATION for cozy temporal graphs beyond the trivial solution, we will need to find exponential time maintenance algorithms that still beat the best current static algorithms. Then again, this can only be succesful if the fastest current algorithm for the corresponding problem is constructive, since our maintenance algorithm needs to be assisted by a constructive algorithm in the first step.

Even if this is possible, we will only have a solution for cozy temporal graphs. Temporal graphs are not always cozy, but we can transform a temporal graph into a cozy temporal graph.

Definition 8.4. *Let $\mathcal{G} = (G_1, \cdots, G_\tau)$ be a temporal graph and $t < \tau$ a time step. Let $H_t^1, \cdots, H_t^k$ be a sequence of ordered neighbour graphs such that $H_t^1 = G_t$ and such that H_t^k is a neighbour graph of G_{t+1}. Then $H_t^1, \cdots, H_t^k$ is a* ***bridge*** *for G_t and G_{t+1}.*

A bridge is called **minimum** if it contains the minimum number of layers required to create a bridge between G_t and G_{t+1}. If G_t and G_{t+1} are already neighbours, a minimum bridge will only consist of G_t. Using bridges, we can transform any temporal graph into a cozy one.

Definition 8.5. *Let $\mathcal{G} = (G_1, \cdots, G_\tau)$ be a temporal graph. For every time step $t < \tau$, let $H_t^1, \cdots, H_t^{k_t}$ be a minimum bridge for G_t and G_{t+1}. Then*

$$\mathcal{G}' = (H_1^1, \cdots, H_1^{k_1}, H_2^1, \cdots, H_2^{k_2}, \cdots\cdots, H_{\tau-1}^1, \cdots, H_{\tau-1}^{k_{\tau-1}}, G_\tau)$$

is a ***cozy transformation*** *for $\mathcal{G}$.*

Solving EVOLVING DOMINATION on a cozy transformation of $\mathcal{G}$ will also solve it on $\mathcal{G}$. Of course, the lifetime of a cozy transformation may be much longer than τ, so if we used a maintenance algorithm on a cozy transformation, we may be required to perform the maintenance algorithm many times. To remedy this, it would be possible to first change the order of the layers in $\mathcal{G}$ such that the sum of the lengths of the minimum bridges between the layers would be minimized to obtain $\mathcal{G}'$ before creating a cozy transformation $\mathcal{G}''$ of that. Solving EVOLVING DOMINATION on $\mathcal{G}''$ will also solve it on $\mathcal{G}$. For simplicity, assume that the layers of any temporal graph mentioned in this section are already in such an optimal order. Let us use σ to denote the lifetime of a cozy transformation of $\mathcal{G}$. Even if we did find a maintenance algorithm B_R that takes $\mathcal{O}(S_R(X))$ time for R-DOMINATION, and is faster than the fastest current algorithm A'_R, which solves it in $\mathcal{O}(T'_R(X))$ time, and assuming that A'_R is indeed constructive, it would take $\mathcal{O}(T'_R(X_1) + \sigma S_R(X))$ time to solve EVOLVING DOMINATION in this way. However, there is another way to do it, without needing A'_R to kickstart our algorithm.

Definition 8.6. *Let $\mathcal{G} = (G_1, \cdots, G_\tau)$ be a temporal graph with vertex set V and let $G_\emptyset = (V, \emptyset)$. Let $\mathcal{G}_\emptyset$ be a temporal graph that includes $G_\emptyset$ as a layer in addition to all the layers of $\mathcal{G}$ and these layers are in an order such that a cozy transformation of $\mathcal{G}_\emptyset$ has a minimum lifetime. The cozy transformation of $\mathcal{G}_\emptyset$ is called a* ***nice cozy transformation*** *of $\mathcal{G}$.*

A nice cozy transformation includes a layer without edges. Computing a solution to R-DOMINATION on this layer is trivial. We can use this solution as input for the maintenance algorithm. This allows us to start at layer $G_\emptyset$ and use the maintenance algorithm to compute solutions for the neighbours of $G_\emptyset$ in the nice cozy transformation and so on. Let ρ denote the lifetime of a nice cozy transformation of $\mathcal{G}$, and let B_R be a maintenance algorithm that takes $\mathcal{O}(S_R(X))$ time for input X, then we can solve EVOLVING DOMINATION on $\mathcal{G}$ in $\mathcal{O}(\rho S_R(X))$ time where X is the input corresponding to the layer that takes the longest for B_R to process. The size of a bridge can not be larger than m, since there are m edges at most that have to be changed to transform a layer G_t into its successor G_{t+1}. Therefore we know that $\rho \leq m(\tau + 1)$. This makes the required running time of our method $\mathcal{O}(m\tau S_R(X))$. Therefore, this method is an improvement over the trival solution if $mS_R(X) < T'_R(X)$.

Theorem 8.2. *Let B_R be a maintenance algorithm for* R-Domination *that runs in* $\mathcal{O}(S_R(X))$ *time for input* X*, then we can solve* R-Domination *in* $\mathcal{O}(mS_R(X))$ *time.*

Proof of Theorem 8.2 Let $G = (V, E)$ be a static graph, $e_1, \cdots, e_m$ the edges of E and let $G_0, \cdots, G_m$ be a sequence of neighbour graphs such that $G_i = (V, \{e_1, \cdots, e_i\})$. The solution to R-Domination on G_0 is trivial. Now, by using B_R m times, we can create a solution to G. Therefore, we can solve R-Domination on G in $\mathcal{O}(mS_R(X))$ time.

□

This is no proof that such a maintenance algorithm can not exist, rather it shows that finding one would mean finding an algorithm that solves R-Domination faster than the current fastest algorithm. This is beyond the scope of this thesis.

8.2 Using the Trivial Solution

Leaving maintenance algorithms behind (for now), we are back to using the trivial solution. We will once again refer back to the fastest algorithms for Dominating Set and (Weak) Roman Domination, which are stated in Section 5. Using the algorithm by Iwata [23] for Dominating Set, the one by Shi and Koh [28] for Roman Domination and the one by Chapelle et al. [10] for Weak Roman Domination, we obtain the following results.

Theorem 8.3. Evolving Dominating Set *can be solved in* $\mathcal{O}(\tau 1.4689^n)$ *and space.*

Theorem 8.4. Evolving Roman Domination *can be solved in* $\mathcal{O}(\tau 1.5014^n)$ *and exponential space.*

Theorem 8.5. Evolving Weak Roman Domination *can be solved in* $\mathcal{O}^*(\tau 2^n)$ *and exponential space.*[3]

9 k-Fold Domination

k-Fold Domination is a problem that is a natural follow up to Temporary Domination and Permanent Domination. To our knowledge, the problem has not been covered previously.

9.1 k-Fold Dominating Set

In this section we will cover the Dominating Set variant of k-Fold Domination. The algorithm we present will be very similar to 1. The algorithm for the Roman Domination variant will use the same concepts introduced here.

Problem 9.1. **k-Fold Dominating Set***: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a positive integer $k \leq \tau$, find the size of minimum k-fold sdf.*

To solve k-Fold Domination for the Dominating Set and Roman Domination domination rules, we can adapt the parameterized algorithm from Section 6. We only need to change Algorithm 1 in the following ways:

- The definition for a partial solution f needs to be changed such that f dominates every vertex $v \in V_i/X_i$ k-fold in $\mathcal{G}[V_i]$, instead of permanently dominating them.
- In the forget nodes, we should allow partial solutions that staisfy temporal colourings c such that the forgotten vertex x is coloured 0 on at least k time steps.

The definition for (satisfying) temporal colourings remains the same as in Section 6.1. The table entry $A_i(c, \kappa)$ for node i of $\mathcal{X}$ still stores the number of partial solutions of size κ on i that satisfy temporal colouring c, although the definition of partial solution is now different. Nodes i of other types will be processed in the same way as in Algorithm 1, and we will show that this will always result in a correct table A_i. The new formal definition for partial solutions is as follows:

[3] The $\mathcal{O}^*$ notation here means that there may be another polynomial factor in n.

Definition 9.1. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X} = (\{X_1, \cdots, X_k\}, X)$ a nice tree decomposition of the underlying graph $G_\downarrow$, i a node of $\mathcal{X}$ and $k \leq \tau$ a positive integer. A* ***partial solution*** *of node i is a domination function $f : V_i \to \mathbb{N}_0$ such that f dominates every vertex $v \in V_i / X_i$ in $\mathcal{G}[V_i]$ k-fold.*

To handle the forget nodes, we need to define the *indicator function* $\mathbb{I}_k : \left\{?, \hat{0}\right\}^\tau \to \{1, 0\}$ such that:

$$\mathbb{I}_k(c_x) = \begin{cases} 1 & \text{if } \#_{\hat{0}}(c_x) > \tau - k \\ 0 & \text{otherwise.} \end{cases}.$$

Algorithm 3: k-Fold Dominating Set

```
FoldDom(temporal graph 𝒢, nice tree decomposition 𝒳 of G↓, positive integer k ≤ τ):
    foreach node i of 𝒳 in BFS manner from the leaves to the root r do
        Let A_i be a table with an entry for each colouring c of X_i and value 1 ≤ κ ≤ n;
        if i is a leaf node then
            foreach colouring c of X_i do
                for κ = 1 to n do
                    if c = (1) and κ = 1 then
                        A_i(c, κ) = 1;
                    else
                        A_i(c, κ) = 0;
        else if i is a forget node then
            Let j be the child of i;
            Let x be the forgotten vertex;
            foreach colouring c of X_i do
                for κ = 1 to n do
                    A_i(c, κ) = A_j(c × {1}, κ) + A_j(c × {?}, κ) −
                      Σ_{c_x ∈ {?,0̂}^τ} 𝕀_k(c_x)(−1)^{#_0̂(c_x)+τ−k} A_j(c × {c_x}, κ);
        else if i is an introduce node then
            Let j be the child of i;
            Let x be the introduced vertex;
            foreach colouring c of X_j do
                for κ = 1 to n do
                    foreach temporal colour c_x do
                        if c_x = 1 then
                            if κ = 0 then
                                A_i(c × {c_x}, κ) = 0;
                            else if there is a vertex x_{i_k} ∈ N_t(x) ∩ X_i such that c_k^t = 0̂ then
                                A_i(c × {c_x}, κ) = 0;
                            else
                                A_i(c × {c_x}, κ) = A_j(c, κ − 1);
                        else if c_x^t = 0̂ and there is a vertex x_{i_k} ∈ N_t(x) ∩ X_i such that c_k = 1
                         then
                            A_i(c × {c_x}, κ) = 0;
                        else
                            A_i(c × {c_x}, κ) = A_j(c, κ);
        else
            Let j and k be the children of i;
            foreach colouring c of X_i do
                for κ = 1 to n do
                    A_i(c, κ) = Σ_{κ_l+κ_r=κ+#_1(c)} A_j(c, κ_l) · A_k(c, κ_r);
    for κ = 1 to n do
        if A(∅, κ) > 0 then
            return κ;
```

Leaf Nodes Leaf nodes are handled in the same way as in Algorithm 1. The proof given for Lemma 6.7 still holds for leaf nodes i in Algorithm 3 because X_i only has one vertex that does not need to be dominated by a partial solution. Therefore a partial solution on a leaf node for the permanent variant is also a partial solution on a leaf node for the k-fold variant, and vice versa.

Forget Nodes Let i be a forget node with bag $X_i = \left(x_{i_1}, \cdots, x_{i_{n_i}}\right)$ and whose child j has bag $X_j = \left(x_{i_1}, \cdots, x_{i_{n_i}}, x\right)$ where x is the forgotten vertex. Then, for k-Fold Dominating Set, we let

$$A_i(c, \kappa) = A_j(c \times \{1\}, \kappa) + A_j(c \times \{?\}, \kappa) - \sum_{c_x \in \{?, \hat{0}\}^\tau} \mathbb{I}_k(c_x)(-1)^{\#_{\hat{0}}(c_x)+\tau-k} A_j\left(c \times \{c_x\}, \kappa\right).$$

This formula ensures that we only disregard partial solutions for which x has less than k dominated vertex appearances. We use the indicator function $\mathbb{I}_k$ to make sure only temporal colours c_x that correlate to such partial solutions are taken into account in the sum. We use the exponent $\#_{\hat{0}}(c_x)+\tau-k$ for (-1) to make sure the partial solutions correlating to temporal colourings with exactly one instance of $\hat{0}$ too many are subtracted. After all, for a temporal colour c_x with $\#_{\hat{0}}(c_x) = \tau - k + 1$ (there is at least one $\hat{0}$ too many) we have that $\#_{\hat{0}}(c_x) + \tau - k = \tau - k + 1 + \tau - k = 2(\tau - k) + 1$ is an odd number. We use the inclusion-exclusion formula to make sure we do not subtract certain partial solutions too often.

Lemma 9.1. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$, $k \leq \tau$ a positive integer, and i a forget node of $\mathcal{X}$ with child j. Assume that A_j is correct. Then Algorithm 3 correctly calculates the table A_i in $\mathcal{O}\left(n(2^\tau + 1)^{n_j}\right)$ time.*

Proof of Lemma 9.1. Let $X_i = \left(x_{i_1}, \cdots, x_{i_{n_i}}\right)$ and let $X_j = \left(x_{i_1}, \cdots, x_{i_{n_i}}, x\right)$ where x is the forgotten vertex. It is clear that partial solutions f on j that satisfy $c \times \{1\}$ also satisfy c on i. Partial solutions f on j that satisfy $c \times \{c_x\}$ for some temporal colour $c_x \in \left\{0, \hat{0}\right\}^\tau$ satisfy c on i only if $\#_{\hat{0}}(c_x) \leq \tau - k$.

For any node k of $\mathcal{X}$, let B_k be a table such that $B_k(c, \kappa)$ is the set of partial solutions on k that satisfy c of size κ. By arguments similar to those in the proof of Lemma 6.5, we know that $B_j(c \times \{?\}, \kappa) / \bigcup_{\#_{\hat{0}}(c_x) > \tau - k} B_j(c \times \{c_x\}, \kappa)$ is the set of partial solutions f with $f(x) = 0$ that satisfy c on i.

For each time step $t \in T$, let C_t be the temporal colour for x such that coordinate t has value $\hat{0}$ and all other coordinates have value ?. $C_t^s = \begin{cases} \hat{0} & \text{if } s = t \\ ? & \text{otherwise} \end{cases}$. Now let $B_i^t = B_i(c \times \{C_t\}, \kappa)$. Note that for any positive integer $y \leq \tau$, a partial solution f that is included in $B_i^{t_1} \cap \cdots \cap B_i^{t_y}$ with $t_a \neq t_b$ for each pair of distinct indices $a \neq b$ also satisfies the temporal colouring $c \times \{C\}$ where coordinates $t_1, \cdots, t_y$ of C have value $\hat{0}$ and all the others have value ?. Conversely, a partial solution f that satisfies temporal colouring $c \times \{C\}$ where coordinates $t_1, \cdots, t_y$ of C have colour $\hat{0}$ and the others have colour ?, must be included in $B_i^{t_1} \cap \cdots \cap B_i^{t_y}$. Therefore for any positive integer $y \leq \tau$, we have that

$$\sum_{1 \leq t_1 < \cdots < t_y \leq \tau} |B_i^{t_1} \cap \cdots \cap B_i^{t_y}| = \sum_{C \in \left\{?, \hat{0}\right\}^\tau, \#_{\hat{0}}(C) = y} |B_i(c \times \{C\}, \kappa)|.$$

Finally, if $\#_{\hat{0}}(c_x) = \tau - k + 1$, then $\#_{\hat{0}}(c_x) + \tau - k$ is an odd number. This allows us to use the following equation.

$$
\begin{aligned}
A_i(c,\kappa) &= A_j(c\times\{1\},\kappa) + |B_j(c\times\{?\},\kappa)/\bigcup_{\#_{\hat{0}}(c_x)>\tau-k} B_j(c\times\{c_x\},\kappa)| \\
&= A_j(c\times\{1\},\kappa) + |B_j(c\times\{?\},\kappa)| - |\bigcup_{\#_{\hat{0}}(c_x)>\tau-k} B_j(c\times\{c_x\},\kappa)| \\
&= A_j(c\times\{1\},\kappa) + A_j(c\times\{?\},\kappa) - |\bigcup_{\#_{\hat{0}}(c_x)>\tau-k} B_j(c\times\{c_x\},\kappa)| \\
&= A_j(c\times\{1\},\kappa) + A_j(c\times\{?\},\kappa) - |\bigcup_{\#_{\hat{0}}(c_x)=\tau-k+1} B_j(c\times\{c_x\},\kappa)| \text{ (By Lemma 6.4.)} \\
&= A_j(c\times\{1\},\kappa) + A_j(c\times\{?\},\kappa) - |\bigcup_{1\le t_1<\cdots<t_{\tau-k+1}\le\tau} B_j^{t_1}\cap\cdots\cap B_j^{t_{\tau-k+1}}| \\
&= A_j(c\times\{1\},\kappa) + A_j(c\times\{?\},\kappa) - \sum_{1\le t_1<\cdots<t_{\tau-k+1}\le\tau} |B_j^{t_1}\cap\cdots\cap B_j^{t_{\tau-k+1}}| \text{ (By the inclusion-exclusion princip} \\
&\qquad + \sum_{1\le t_1<\cdots<t_{\tau-k+2}\le\tau} |B_j^{t_1}\cap\cdots\cap B_j^{t_{\tau-k+2}}| \\
&\qquad - \cdots \\
&\qquad + (-1)^\tau |B_j^1\cap\cdots\cap B_j^\tau| \\
&= A_j(c\times\{1\},\kappa) + A_j(c\times\{?\},\kappa) + \sum_{c_x\in\{?,\hat{0}\}^\tau} \mathbb{I}_k(c_x)(-1)^{\#_{\hat{0}}(c_x)+\tau-k} |B_j(c\times\{c_x\},\kappa)|. \\
&= A_j(c\times\{1\},\kappa) + A_j(c\times\{?\},\kappa) + \sum_{c_x\in\{?,\hat{0}\}^\tau} \mathbb{I}_k(c_x)(-1)^{\#_{\hat{0}}(c_x)+\tau-k} A_j(c\times\{c_x\},\kappa).
\end{aligned}
$$

To calculate A_i, we need to look up $n(2^\tau+1)^{n_j}$ table entries, so the required time is $\mathcal{O}\left(n(2^\tau+1)^{n_j}\right)$.
□

Introduce Nodes Introduce nodes i are handled by extending a partial solution on the child node j by adding a single vertex x to the domain. This new vertex cannot have neighbours outside X_i. Therefore the value the extended partial solution assigns to this new vertex cannot cause a vertex outside X_i to become dominated. Moreover, the new vertex does not need to be dominated itself, therefore the different definition for *partial solutions* brings no complications for the way we handle introduce nodes, so Lemma 6.9 still holds.

Join Nodes Finally, join nodes i are handled by taking pairs of partial solutions for the two children j and k that both satisfy the same colouring, and creating a new fused partial solution. Since the vertices in V_j/X_i do not interact in any way with the vertices in V_k/X_i and we assume they are both already dominated, Lemma 6.10 still holds.

Termination The root node is handled in the same way as in Algorithm 1.

Theorem 9.1. *Algorithm 3 solves k-Fold Dominating Set correctly.*

Proof of Theorem 9.1. This is easily proven by repeating the proof of Theorem 6.1, although we have to replace the use of Lemma 6.8 with Lemma 9.1.
□

Theorem 9.2. *k-Fold Dominating Set can be solved in $\mathcal{O}\left(n^3\left(2^\tau+1\right)^{\ell+1}\right)$ time and $\mathcal{O}\left(n\left(2^\tau+1\right)^{\ell+1}\right)$ space.*

Proof of Theorem 9.2. Since the bottleneck for the running time of this algorithm is still the processing of the join nodes, the proof for this theorem is the same as for Theorem 6.2.

□

9.2 k-Fold Roman Domination

k-Fold Roman Domination can be solved by combining the new techniques introduced in the previous subsection with the technique from Section 6.3.

Problem 9.2. **k-Fold Roman Domination***: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a positive integer $k \leq \tau$, find the size of minimum k-fold rdf.*

To take into account the vertices that get label 1, we use the technique introduced in Section 6.3. For any node i of $\mathcal{X}$, we ignore the colour 1 for any vertex in X_i and only start taking this possibility

into account when we forget the vertex.

Algorithm 4: k-Fold Roman Domination

```
FoldRom(temporal graph 𝒢, nice tree decomposition 𝒳 of G↓, positive integer k ≤ τ):
  foreach node i of 𝒳 in BFS manner from the leaves to the root r do
    Let A_i be a table with an entry for each colouring c of X_i and value 1 ≤ κ ≤ n;
    if i is a leaf node then
      foreach colouring c of X_i do
        for κ = 1 to n do
          if c = (2) and κ = 2 then
            A_i(c, κ) = 1;
          else
            A_i(c, κ) = 0;
    else if i is a forget node then
      Let j be the child of i;
      Let x be the forgotten vertex;
      foreach colouring c of X_i do
        for κ = 1 to n do
          A_i(c, κ) = A_j(c × {2}, κ) + A_j(c × {?}, κ) −
            Σ_{c_x ∈ {?, 0̂}^τ} 𝕀_k(c_x)(−1)^{#_0̂(c_x)+τ−k} A_j(c × {c_x}, κ) + A_j(c × {?}, κ − 1);
    else if i is an introduce node then
      Let j be the child of i;
      Let x be the introduced vertex;
      foreach colouring c of X_j do
        for κ = 1 to n do
          foreach temporal colour c_x do
            if c_x = 2 then
              if κ < 2 then
                A_i(c × {c_x}, κ) = 0;
              else if there is a vertex x_{i_k} ∈ N_t(x) ∩ X_i such that c_k^t = 0̂ then
                A_i(c × {c_x}, κ) = 0;
              else
                A_i(c × {c_x}, κ) = A_j(c, κ − 2);
            else if c_x^t = 0̂ and there is a vertex x_{i_k} ∈ N_t(x) ∩ X_i such that c_k = 2
             then
              A_i(c × {c_x}, κ) = 0;
            else
              A_i(c × {c_x}, κ) = A_j(c, κ);
    else
      Let j and k be the children of i;
      foreach colouring c of X_i do
        for κ = 1 to n do
          A_i(c, κ) = Σ_{κ_l+κ_r=κ+2#_2(c)} A_j(c, κ_l) · A_k(c, κ_r);
  for κ = 1 to n do
    if A(∅, κ) > 0 then
      return κ;
```

Leaf Nodes, Introduce Nodes and Join Nodes

Lemma 9.2. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$, $k \le \tau$ a positive integer, and i a leaf node, introduce node or forget node of $\mathcal{X}$ with child j (and k in case of a*

join node). Assume that A_j (and A_k) is correct. Then Algorithm 4 correctly calculates the table A_i.

Proof of Lemma 9.2. Leaf, introduce and join nodes are handled in the same way as in Algorithm 2, since it does not matter in how many time steps they are dominated in such nodes. Therefore, the proofs for Lemmas 6.11, 6.13 and 6.14 are sufficient to prove that these nodes are also handled correctly here. Therefore, if i is any of those types of nodes, and its children (if they have any) have correct tables, then A_i will be calculated correctly by Algorithm 4.

□

Forget Nodes Let i be a forget node with bag $X_i = \left(x_{i_1}, \cdots, x_{i_{n_i}}\right)$ and whose child j has bag $X_j = \left(x_{i_1}, \cdots, x_{i_{n_i}}, x\right)$ where x is the forgotten vertex. Then the table is calculated as follows:

$$A_i(c,\kappa) = A_j(c\times\{2\},\kappa)+A_j(c\times\{?\},\kappa)-\sum_{c_x\in\{?,\hat{0}\}^\tau} \mathbb{I}_k(c_x)(-1)^{\#_{\hat{0}}(c_x)+\tau-k+2}A_j\left(c\times\{c_x\},\kappa\right)+A_j(c\times\{?\},\kappa-1).$$

Lemma 9.3. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, $\mathcal{X}$ a nice tree decomposition of $G_\downarrow$, $k \leq \tau$ a positive integer, and i a forget node of $\mathcal{X}$ with child j. Assume that A_j is correct. Then Algorithm 4 correctly calculates the table A_i.*

Proof of Lemma 9.3. Let i be a forget node of $\mathcal{X}$ with child j. We know that

$$A_i(c,\kappa) = A_j(c\times\{2\},\kappa) + A_j(c\times\{1\},\kappa) + A_j(c\times\{0\},\kappa).$$

In the proofs of Lemma 9.1 and Theorem 6.3 we have already shown that the way Algorithm 4 calculates $A_j(c\times\{0\},\kappa)$ and $A_j(c\times\{1\},\kappa)$ respectively are correct. Therefore we know that the forget nodes are calculated correctly by Algorithm 4 if the child node j of i already has a correct table.

□

Theorem 9.3. *Algorithm 4 solves k-Fold Roman Domination correctly.*

Proof of Theorem 9.3. Lemmas 9.2 and 9.3 show that all nodes are calculated correctly. This, in combination with the proof for Theorem 9.1 is enough to proof this theorem.

□

Theorem 9.4. *k-Fold Roman Domination can be solved in $\mathcal{O}\left(n^3\left(2^\tau+1\right)^{\ell+1}\right)$ time and $\mathcal{O}\left(n\left(2^\tau+1\right)^{\ell+1}\right)$ space.*

Proof of Theorem 9.4. All node types take the same amount of time as in Algorithm 3. Since the bottleneck for the running time of this algorithm is still the processing of the join nodes, the proof for this theorem is the same as for Theorem 6.2.

□

10 Marching Domination

The Marching Domination problem is similar to Permanent Domination in the sense that every vertex needs to be dominated on every time step. However in this problem, the "armies" are allowed to move every at the end of each time step t. They are, however, limited to using only edges

that exist in layer G_t. Of course, they do not *have* to move. A solution where every vertex receives one army which does not move during the entire lifetime of the graph is a valid solution for all three variants of the problem, therefore we can put an upper bound of n on the number of armies k.

In order to solve the minimization problem which minimizes the number of armies used, we first solve the decision problem k-Marching Domination where the desired number of armies k is already set and we need to figure out whether or not a marching domination of the given size k exists. We will provide an algorithm which uses a combination of branching and dynamic programming techniques. Rather than providing three separate algorithms for the three problems, our algorithm works for all three variants of k-Marching Domination. This algorithm can then be used to solve the minimization problems.

Problem 10.1. **k-Marching Dominating Set***: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a positive integer k, decide whether an R_{Dom}-marching domination of size k exists.*

Problem 10.2. **k-Marching Roman Domination***: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a positive integer k, decide whether an R_{Rom}-marching domination of size k exists.*

Problem 10.3. **k-Marching Weak Roman Domination***: Given a temporal graph $\mathcal{G} = (V, E, \tau, \lambda)$ and a positive integer k, decide whether an R_{Weak}-marching domination of size k exists.*

The dynamic programming part of our algorithm relies on the following: If we know that there is a strategy s of size k that dominates the first $t-1$ layers, we know the position of the armies on time step $t-1$ and we know that moving the armies in a certain way will allow s to dominate layer G_t, then we know there is a strategy of size k that dominates the first t layers. To formalize this, we introduce the notion of a *t-partial solution.*

Definition 10.1. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and t a time step. A **t-partial solution** is a marching domination on the temporal graph $\mathcal{G}^t = (G_1, \cdots, G_t)$.*

In each step of the algorithm, we will attempt to move the armies around the graph. We will use the notion of *intermediate functions* from Section 4.8. We will also use a slightly altered version of *partial and valid moves* from that section.

Definition 10.2. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, t a time step and $f : V \to \mathbb{N}_0$ a domination function on G_t using k armies. Let $g : V \to \mathbb{N}_0^2$ be an intermediate function on G_t. If $\{u, v\} \in E_t$ and $g(u).x > 0$, or if $u = v$, then the intermediate function $g_{u \to v}$ is called a **partial move** for g on time step t. Now let $g_0, \cdots, g_k$ be a sequence of k intermediate functions on G_t such that $g_0(v).x = f(v)$ for every vertex $v \in V$ and g_i is a partial move for g_{i-1} on time step t for $1 \leq i \leq k$. Let $f'(v) = g_k(v).x + g_k(v).y$ for all vertices $v \in V$. Then f' is a **valid move** for f on time step t.*

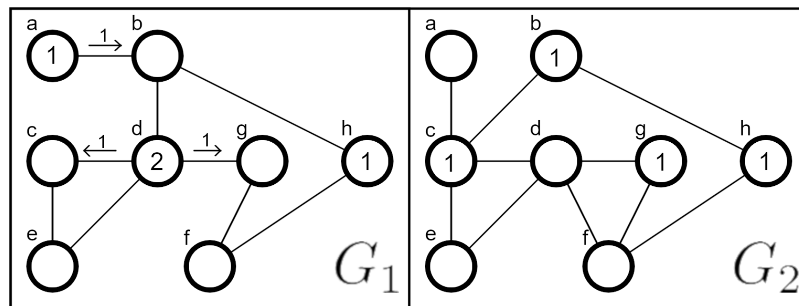

Figure 10.1: A temporal graph with $\tau = 2$. Each individual arrow in the first picture each represents a partial move. Together, they form a valid move. The second picture shows the result of the valid move for the next time step.

A valid move is a possible way to move the armies using the edges of layer G_t. Note that a sequence of k intermediate functions such as in Definition 10.2 can move any army at most once, since $g_{u \to v}$ decreases the x coordinate of its label for u compared to g, and increases only the y coordinate of its label for v compared to g. Since k movements are performed, every army can either move over one edge, or remain on its current vertex.

10.1 Positive Instance Based Dynamic Programming

We create a table A. Let $f : V \to \mathbb{N}_0$ be some domination function and t some time step. Then table entry $A(f, t)$ contains the value `true` if there exists a t-partial solution s on $\mathcal{G}$ such that $s_t = f$ which still has a chance to lead to a solution. It contains the value `false` if no t-partial solution s exists such that $s_t = f$, or if all such t-partial solutions have been confirmed not to lead to a solution. Finally, it contains the value `unexplored` if the algorithm has not tried this domination function yet on this time step. This type of algorithm is called a *positive instance based dynamic programming algorithm* because we use dynamic programming, but only explore table entries which can still lead to a correct solution.

The input of the algorithm is the temporal graph, the value of k and a domination rule R. The algorithm starts by exploring the first unexplored entry $A(f, 1)$ in the first column relating to time step 1. Whenever we *explore* a previously unexplored table entry $A(f, t)$, we first check if f is an R-domination function on layer G_t. If f is an R-domination function on layer G_t, the value of $A(f, t)$ is changed to `true`, and we can continue exploring this possibility. We will then choose a valid move f' for f and start exploring $A(f', t + 1)$. If f is not an R-domination function on layer G_t, the value of $A(f, t)$ is changed to `false`. If $A(f, t) =$ `true` but we have $A(f', t + 1) =$ `false` for every possible valid move f' for f, then we know that this possibility can never lead to a solution, and thus the value of $A(f, t)$ is changed to `false`.

Whenever we set a table entry to `false`, or whenever we start exploring a valid move that leads to a table entry that is already set to `false`, we stop exploring this table entry and return to the last table entry that we encountered with the value `true` or, if no such table entry exists, to the next unexplored entry in the first column of the table. Therefore the algorithms order of exploring table entries is reminiscent of depth-first search.

If all entries in the first column of the table are set to `false`, the algorithm returns `false`, meaning that no marching domination of size k exists on $\mathcal{G}$. If we encounter a table entry $A(f, \tau)$ which is set to `true`, the algorithm returns `true`, meaning that there exists a marching domination of size k on $\mathcal{G}$.

Algorithm 5: MARCHING DOMINATION

```
March(temporal graph 𝒢, domination rule R):
    for integer k = 1 to n do
        if Exists(k) then
            return k;

Exists(integer k):
    foreach domination function f on G_1 of size k do
        Explore(f, 1);
        if A(f, 1)==true;
         then
            return true;
    return false;

Explore(domination function f, time step t):
    if f is an R-domination function on G_t then
        if t==τ then
            A(f, τ) = true;
            return;
        else
            foreach valid move f' for f on G_t do
                if A(f', t + 1)==unexplored then
                    Explore(f', t + 1);
                    if A(f', t + 1)==true then
                        A(f, t) = true;
                        return;
    A(f, t) = false;
    return;
```

Theorem 10.1. *Algorithm 5 correctly solves the* MARCHING DOMINATION *problem.*

Proof of Theorem 10.1. Let us first prove that the `Exists` algorithm is correct. Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, R a domination rule and $k \leq n$ a positive integer. We have to prove that if there is an R-marching domination of size k on $\mathcal{G}$, then `Exists(`k`)` returns `true`, and it returns `false` otherwise.

First, let us assume that such a marching domination $s = \{a_1, \cdots, a_k\}$ exists. We know that for each time step t, the domination function s_t is an R-domination function on layer G_t. Therefore, if the algorithm reaches the entry $A(s_1, 1)$, it will find that s_1 is an R-domination function on G_1, and will start exploring valid moves for s_1 on time step 1.

Let $t < \tau$ be some time step. Assume the algorithm is exploring all valid moves for s_t on some time step $t < \tau$. The algorithm may terminate before exploring $A(s_{t+1}, t+1)$, in which case it will return `true`. If it does reach $A(s_{t+1}, t+1)$, it will find that s_{t+1} is an R-domination function on time step $t+1$ and thus either terminates (if $t + 1 = \tau$) or starts exploring all valid moves for s_{t+1} on time step $t+1$. Through induction, we find that `Exists(`k`)` will eventually terminate with value `true`: either by finding $A(s_\tau, \tau)$, or by some route it finds earlier.

Now assume that such a marching domination does not exist. For contradiction, assume that `Exists(`k`)` returns `true`. For that to happen, there must be a domination function s_τ such that $A(s_\tau, \tau) =$ `true`. If the algorithm explores a table entry $A(s_t, t)$ for some time step $t > 1$, s_t must be a valid move for some domination function s_{t-1} on time step t, which is an R-domination function for

G_t. Through induction we then know there must be a sequence $(s_1, \cdots, s_\tau)$ of domination functions such that $A(s_t, t) = \texttt{true}$ for every time step t and s_{t+1} is a valid move for s_t on time step t for every time step $t < \tau$. Then there must be an R-marching domination $s = \{a_1, \cdots, a_k\}$ which induces the domination functions $s_1, \cdots, s_\tau$, which is a contradiction. Therefore, we know that `Exists` correctly solves the k-Marching Domination Problem.

We perform `Exists(k)` for each $k \in \{1, \cdots, n\}$. We know that `March` will eventually terminate, because putting one army on each vertex and leaving them there on each time step induces a correct solution of size n. Therefore we know that Algorithm 5 correctly solves Marching Domination.

□

Theorem 10.2. *The k-Marching Domination decision problem can be solved using $\mathcal{O}\left(\binom{n+k-1}{k} nd^{k+1}\tau\right)$ time for R_{Dom} and R_{Rom}, and $\mathcal{O}\left(\binom{n+k-1}{k} nd^{k+3}\tau\right)$ for R_{Weak}, and $\mathcal{O}\left(\binom{n+k-1}{k}\tau\right)$ space.*

Proof of Theorem 10.2. A worst-case scenario is one where every table entry is processed. This happens when every table entry results in a `true`, except those corresponding to the last time step, since setting any one of these entries to `true` could terminate the algorithm before all table entries are processed. A graph that is a clique on every time step except for the last time step, where all edges suddenly disappear would lead to this scenario, as long as $k < n$.

There are k armies and n vertices. A domination function places each of these k armies on another vertex. The armies become indistinguishable as it does not matter which armies are placed on vertex v, only how many. The vertices remain distinguishable. Therefore the number of possible domination functions and therefore the number of table entries in one column is equal to the number of ways we can distribute k indistinguishable objects over n boxes, which is equal to $\binom{n+k-1}{k}$ [27]. Therefore, there are $\binom{n+k-1}{k}\tau$ table entries in the table. This proves the claimed space requirement.

Every table entry is processed in this worst-case scenario, which means we check for each domination function if it is an R-domination function. For $R \in \{R_{\text{Dom}}, R_{\text{Rom}}\}$, this means checking the closed neighbourhood of every vertex, taking $\mathcal{O}(nd)$ time, d being the maximum degree over all layers. For $R = R_{\text{Weak}}$, it is a bit more complicated. If some vertex v has no army stationed on itself and does not have neighbours with two or more armies, but does have at least one neighbour u with one army, then we have to check if v is safely or unsafely defended by u. To check this, we also have to check, for each unsecured neighbour w of u, whether w is *only* defended by u or if it has multiple defenders. If it is only defended by u, and does not neighbour v itself (otherwise v defends w after the army moves there) then w is weakly defended by u and thus u unsafely defends v. Since we have to check all neighbours of v, u and w in this scenario, this takes $\mathcal{O}\left(nd^3\right)$ time. Therefore, in this scenario, we spend $\mathcal{O}\left(\binom{n+k-1}{k} nd\tau\right)$ time on processing table entries for R_{Dom} and R_{Rom}, and $\mathcal{O}\left(\binom{n+k-1}{k} nd^3\tau\right)$ time for R_{Weak}.

In addition to processing the validity of each table entry, we also need to generate all valid moves for each table entry. Many of the domination functions generated in this way, are duplicates of domination functions associated with table entries which have already been checked, and will not be processed again. Nevertheless, generating them is mandatory, and takes time as well. For each of the k armies in a domination function, there are $d + 1$ possible moves: one for each neighbour and one for not moving. This means that each domination function has $(d + 1)^k$ possible valid moves that need to be checked. This leads to a running time of $\mathcal{O}\left(\binom{n+k-1}{k} nd^{k+1}\tau\right)$ for R_{Dom} and R_{Rom}, and $\mathcal{O}\left(\binom{n+k-1}{k} nd^{k+3}\tau\right)$ for R_{Weak}.

□

Theorem 10.3. *The Marching Domination problem can be solved using $\mathcal{O}\left(\binom{2n-1}{n} nd^{n+1}\tau\right)$ time for R_{Dom} and R_{Rom}, and $\mathcal{O}\left(\binom{2n-1}{n} nd^{n+3}\tau\right)$ for R_{Weak}, and $\mathcal{O}\left(\binom{2n-1}{n}\tau\right)$ space.*

Proof of Theorem 10.3. In a worst-case scenario, the answer to k-MARCHING DOMINATION is n. In that case, `Exists` is ran n times. The last time it is ran contributes the largest part of the total running time, and since the big O notation only takes the largest summant into account, the proposed running time complexity is achieved.

□

10.2 Improvements for Practical Scenarios

In this section, we will introduce ideas and techniques to reduce the number of table entries that will be processed. Though the techniques proposed in this section do not improve the theoretical complexity of the algorithm, they are expected to improve the speed of the algorithm in a practical environment.

If we can prove for some domination function property X that, if at least one R-marching domination of size k exists on $\mathcal{G}$, then there must also be at least one R-marching domination of the same size with property X, then we would only have to consider those table entries corresponding with that property. This allows us to skip a (hopefully large) portion of the table entries, thus reducing the required time of the algorithm.

In particular, one area in which our solution can be improved is the number of table entries that have to be considered in the first column. We will prove that if an R-marching domination of size k exists on $\mathcal{G}$, then there must also be an R-marching domination of the same size such that on the first time step, there are no vertices with more than 1 army in the case $R = R_{\text{DOM}}$ or more than 2 armies in the other two cases. To prove this, we will first introduce some definitions.

Definition 10.3. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, t a time step, $v \in V$ some vertex, s an R-marching domination for some domination rule R and $a \in s$ some army from s. If, in $s \setminus \{a\}$, vertex v is not defended on time step t, then army a is* ***necessary*** *for v on time step t. Otherwise a is* ***redundant*** *for v on time step t. If a is not necessary for any vertex on time step t, then a is redundant on time step t. If a is redundant for every time step $t' \geq t$, then a is* ***totally redundant*** *on time step t.*

Some armies are redundant on some time steps and necessary on others. If an army is redundant for one or more consecutive time steps but will be necessary later, then one might say that that army is waiting for its time to become useful. The next time step that the army will become necessary is called the army's current *goal time*, and the vertex that it will occupy on that time step is called its current *goal*.

Definition 10.4. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, t a time step, s a strategy and $a \in s$ an army. If a is not totally redundant on time step t, let $t' \geq t$ be the first time step starting from t on which a is necessary. Then t' is the* ***goal time*** *of army a at time step t. In this case, vertex $a^{t'}$ is the* ***goal*** *of army a at time step t.*

While "waiting" for its goal time, an army might be travelling on some route that leads to its goal. Or it might be that it is already present on its goal and simply sitting still until its goal time. Whatever the case, during this waiting time, an army is not needed to fully dominate the graph, although removing all redundant armies on a certain time step could lead to vertices not being dominated anymore. After all, removing an army could cause another redundant army to suddenly become necessary. Some armies, however, could be removed entirely without losing the marching domination property, as long as we only remove a single army at a time.

Lemma 10.1. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, s an R-marching domination for some domination rule R and $a \in s$ an army. If army a is totally redundant on time step 1, then the strategy $s' = s \setminus \{a\}$ is also an R-marching domination.*

Proof of Lemma 10.1. As per Definition 10.3, removing a will not cause any vertex appearance $v_t \in V^t$ to become undefended, otherwise a would be necessary for v on time step t and would thus not be totally redundant.

□

In the same vein, armies that are not necessary yet at time step 1, can be moved around without losing the marching domination property. Since it does not matter what an army does between the time steps where it is necessary (as long as it reaches its goals at the right time steps) we could move an army to start at its first goal without any danger of ruining our solution.

Lemma 10.2. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, s an R-marching domination for some domination rule R and $a \in s$ an army. Let $v \neq a^1$ be the goal of army a on time step 1 and let the corresponding goal time be t_a. Let $\hat{a}$ be a new army such that*

$$\hat{a}^t = \begin{cases} v & \text{if } t \leq t_a \\ a^t & \text{otherwise} \end{cases} .$$

Then the strategy $s' = s \cup \{\hat{a}\} \setminus \{a\}$ is also an R-marching domination.

This lemma implies that if we change an R-marching domination by moving an army so it starts at its first goal and stays there until its goal time, but behaves the same after that time, does not cause the strategy to lose the R-marching domination property.

Proof of Lemma 10.2. Army a is redundant during the first $t_a - 1$ time steps. Therefore, it does not matter on which vertex it is stationed until time step t_a. Any vertex appearance that is defended by a during or after time step t_a in s, is defended by $\hat{a}$ in s', since $\hat{a}$ behaves the same as a and all other armies behave the same in s' as they did in s. Therefore, any vertex appearance defended in s is also defended in s'.

□

Definition 10.5. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph, t a time step, v a vertex and s an R-marching domination for some domination rule R. If $R = R_{\text{DOM}}$, then v **overflows** on time step t if $s_t(v) > 1$. If $R = R_{\text{ROM}}$ or $R = R_{\text{WEAK}}$, then v overflows on time step t if $s_t(v) > 2$.*

A vertex *overflows* on time step t if there are more armies stationed on it than necessary. In particular, for R_{DOM}, if on some time step t there are multiple armies stationed on vertex v, it will provide the same coverage as a single army would. In some cases it can be necessary for a vertex to overflow, as shown in Figure 4.7. It can even be necessary for $\lceil n/2 \rceil$ armies to be on the same vertex, as shown in the following figure.

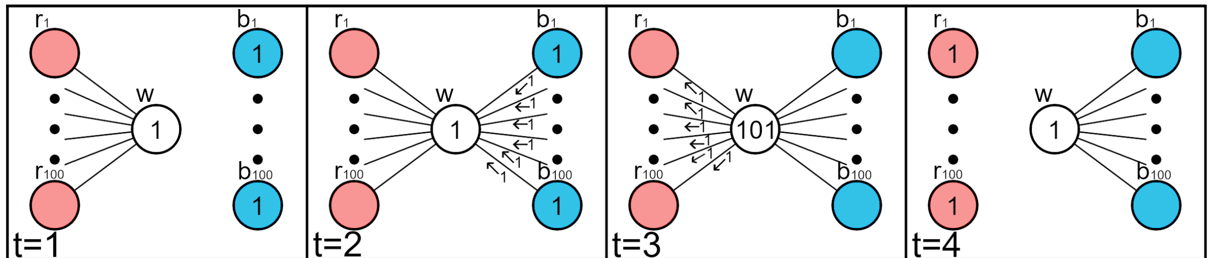

Figure 10.2: A temporal graph with $\tau = 4$ containing 100 red vertices $r_1, \cdots, r_{100}$, 100 blue vertices $b_1, \cdots, b_{100}$ and one white vertex w. The red vertices all start connected to w and can thus be defended by placing a single army on w. The blue vertices all start as isolated vertices and thus need to be defended by placing an army on every blue vertex. On the second time step, the blue armies become connected to w, allowing us to move all the armies currently on blue vertices to w, causing it to overflow. This, in turn, allows us to move 100 armies from w to the red vertices in the third time step, which is necessary because the red vertices become isolated in the last time step.

We now have all definitions and lemmas needed to prove the claim made at the beginning of this section. To do so, we provide an algorithm that changes any solution into another solution of the same size with the required property.

Lemma 10.3. *Let $\mathcal{G} = (V, E, \tau, \lambda)$ be a temporal graph and $k \leq n$ a non-negative integer. If an R-marching domination of size k exists on $\mathcal{G}$, then an R-marching domination s of size k exists such that on the first time step, no vertex overflows.*

Proof of Lemma 10.3. To prove this, we will show that we can transform any R-marching domination s of size k into one that satisfies the requirement. This can be done using an algorithm that changes the strategy in each time step, without losing the property that it is a marching domination.

For each vertex, we will define a property called the *moment*. The moment of a vertex v is 0 if at time step 1 there are no armies stationed on v, or if every army on v at time step 1 will move away from v before becoming necessary, or stay on v until the last time step without ever becoming necessary. If at least one army a which is stationed on v at time step 1 will become necessary before moving away from v (this also implies that v is its first goal), then the moment of v is equal to the goal time of a. Note that there can only be one such army on v if $R = R_{\text{DOM}}$. If there were more than one, for example armies a and b, then let t_a be the goal time of a and t_b be the goal time of b, with $t_a \leq t_b$, then at time step t_a, a and b are both stationed on v, which means a can not be necessary by definition. If $R \neq R_{\text{DOM}}$, then there can be at most two such armies on v, for similar reasons. Our algorithm starts by calculating the moment of each vertex. This is phase 1.

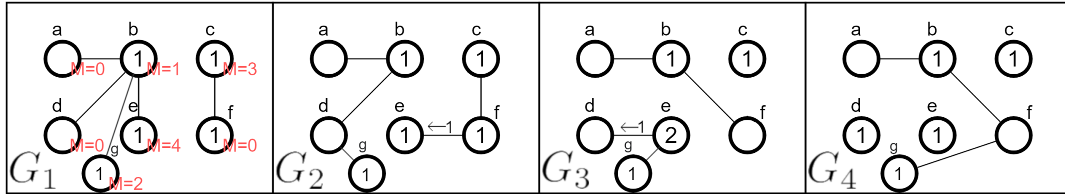

Figure 10.3: A temporal graph with $\tau = 4$ with an R_{DOM}-Marching Domination. In the first time step, each vertex is labelled with its moment. a and d have moment 0 because they have no army on the first time step. b has moment 1 because in the first time step, it contains an army that is necessary for a and d. c has moment 3 because it contains an army that will stay on it until it becomes necessary for c itself in time step 3. e has moment 4 because it contains an army that will stay there until it becomes necessary for e itself in time step 4. f has moment 0 because the only army it contains, moves away from it before becoming necessary. g has moment 2 because it contains an army that stays on it until it becomes necessary for g itself in the second time step.

In the second phase of the algorithm, we will process each army a once in the following way. If a is totally redundant at time step 1, we change the solution s by removing a completely (though we remember a for later). Lemma 10.1 proves this change does not cause the strategy to lose the marching domination property. Removing a might cause another army to become necessary in time steps where it was previously redundant. This is because a vertex appearance $v_t \in V^t$ may have been defended by a and some other army b (and no other armies), both being redundant. Removing a would then cause b to become necessary for defending v_t. Since removing a can cause this to happen, we calculate the moments of all vertices in the graph again after removing a.

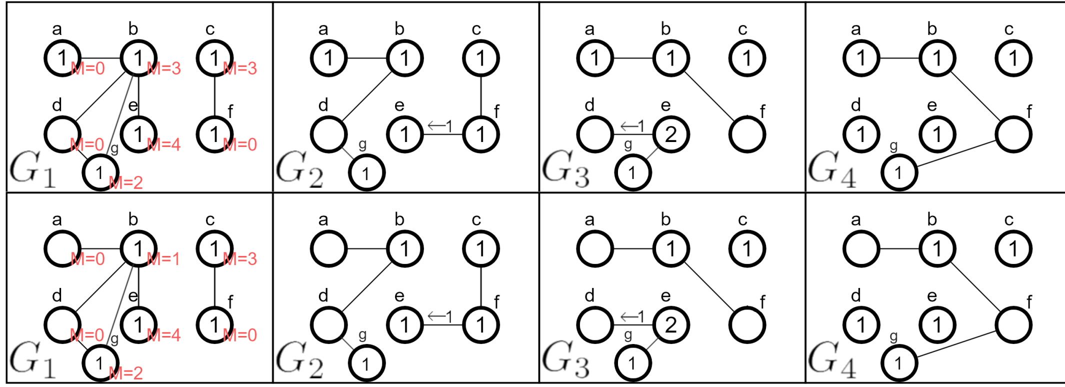

Figure 10.4: A temporal graph similar to the one from Figure 10.3, but with the edge $\{d, g\}$ added in time step 1. In the first pictue, an army is stationed vertex a, which is totally redundant. The moment of b then becomes 3 because it is no longer necessary for a or d in the first two time steps. Removing the army on a results in the second picture, and causes b to become necessary for vertex a in the first time step where it was previously redundant.

If a is not totally redundant, we change the solution s by replacing a by $\hat{a}$ as in Lemma 10.2. The lemma also proves that doing this will not cause the strategy to lose the marching domination property. Like when removing a, moving a to another location for a number of time steps may cause another army to become necessary in time steps where it was previously redundant. In addition, by moving a to some vertex v, the moment of v may increase. It can also stay the same, but only if $R = R_{\text{ROM}}$ or $R = R_{\text{WEAK}}$ and another army was already stationed on v. In any other case, it will increase, because if it did not, it would imply that the current goal of v is $t_b > t_a$ for some other army b that is already stationed on v and stays there until that time, where t_a is the first goal time of a and t_b that of b. But this would imply that on time step t_a, a would not become necessary since b is defending the same vertices as a, which is a contradiction. Therefore, after moving a, we need to calculate the moments of all vertices in the graph again.

Our goal is to create a strategy such that each army is either removed or stationed on its first goal until its goal time is reached. Note that if we apply step 2 of the algorithm to such a strategy, nothing would be changed. However, moving an army a may cause the first goal of another army b to change. It may also make b redundant on certain time steps where it was previously necessary or even cause it to become totally redundant. In other words, after performing step 2, we are not guaranteed to be left with a desirable strategy s yet. Therefore we will repeat step 2 until we do have such a strategy.

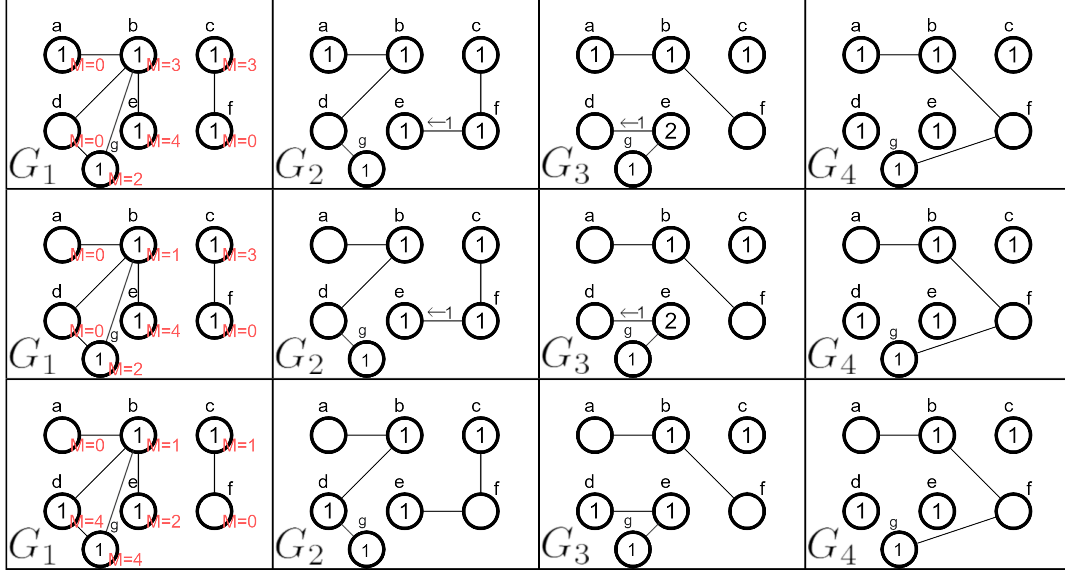

Figure 10.5: Applying this algorithm to the first picture first processed the army on a which is removed since it is totally redundant. This results in the second picture. It then processes the other armies, finding that they do not need to be changed at the moment, until it reaches vertex f. The army on f does not become necessary until it reaches vertex d in the final time step, so the algorithm moves it such that it starts there and stays there until the end of the graph's lifetime, resulting in the third picture. After this move, applying the algorithm again would not move anything, thus the algorithm terminates.

We know that we will always reach such a strategy in a finite number of steps for the following reason: The number of consecutive iterations of step 2 without increasing any moments is bounded by $\frac{n}{2} + k$, because if we assume at least one move or one removal takes place in each iteration of step 2, after this number of iterations without increasing any moments, we may have had at most k removals. After all, removing all k armies would cause the strategy to lose the marching domination property, since nothing is dominated anymore. We also may have had $\frac{n}{2}$ moves where an army was moved to a vertex where another army (which would also become necessary there before moving) is already stationed if $R = R_{\text{Rom}}$ or $R = R_{\text{Weak}}$, since two armies may both be necessary to dominate a vertex. After $\frac{n}{2}$ such moves, we must have $\frac{n}{2}$ vertices with at least 2 armies each, which means $k \geq n$, in which case there are no other armies left to move or remove. Therefore, after each $\frac{n}{2} + k$ iterations, at least one moment must have increased. Since the moments are bounded by τ, and there are n moments, the step 2 loop must terminate after a finite number of iterations.

Once we have a desirable strategy, no vertex can be overflowing. If a vertex v is overflowing in some iteration of step 2, there must be an army a stationed there on time step 1, in addition to one or two (depending on R) armies that will become necessary there and whose goal times define the moment of v. In the next iteration of step 2, a would either have been moved to its first goal (which can not be v) or removed. Step 2 only terminates if nothing changes during an iteration, so this situation must have been resolved before the step terminated.

Since $n \geq k$, there must be at least one empty vertex for each removed army. In step 3 of the algorithm, we place each removed army back on the graph on an empty vertex and make it stay there during the entire life time of the graph. Doing so can obviously not cause the strategy to lose the marching domination property.

After performing all three steps of this algorithm, we have obtained a marching domination of size k such that there are no overflowing vertices on the first time step.

□

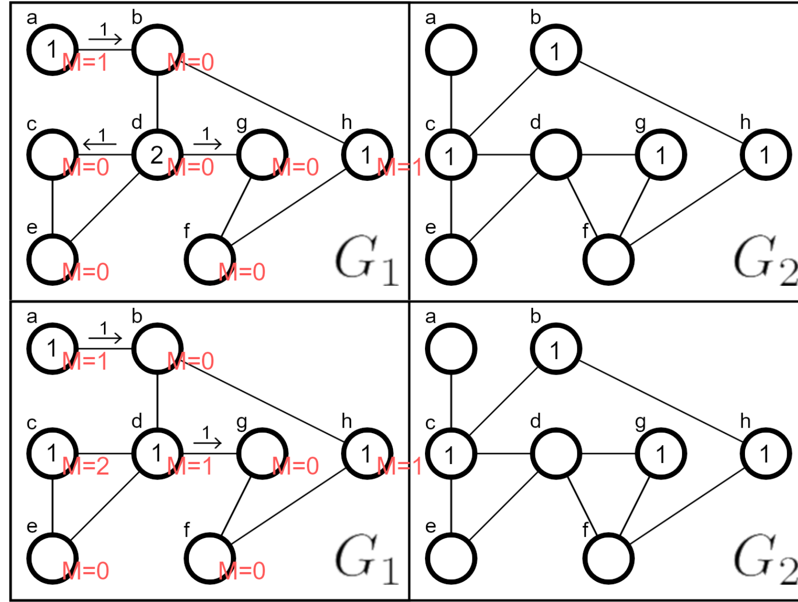

Figure 10.6: A temporal graph with $\tau = 2$ with an R_{DOM}-marching domination. In the first picture, vertex d is overflowing in the first time step. Applying the algorithm to this graph moves one of the armies on d to vertex c on the first time step, creating an R_{DOM}-marching domination of the same size without overflowing vertices on the first time step, shown in the second picture.

Lemma 10.3 shows that Algorithm 5 only needs to consider domination functions that have no overflowing vertices on time step 1. For R_{DOM}, this means that the number of table entries that need to be processed for the first column can be reduced from $\binom{n+k-1}{k}$ to $\binom{n}{k}$. It is trivial to see that this is indeed a smaller number of table entries to process. For R_{ROM} and R_{WEAK}, the number of table entries on the first column that need to be processed, is equal to the number of ways one can arrange k indistinguishable objects into n distinguishable boxes with capacity 2. There are

$$\binom{n+k-3i-1}{n-1}$$

possible ways to arrange k indistinguishable objects into n distinguishable boxes where at least the first i boxes are overflowing. We obtain this number because at least i specified vertices already have at least 3 armies, decreasing the number of armies to distribute by $3i$, the formula then follows from [27]. Note that $\binom{n+k-1}{k} = \binom{n+k-1}{n-1}$, so if $i = 0$, we have our original formula without capacity. If we know that at least i vertices are overflowing but we do not specify which vertices, then there are $\binom{n}{i}$ ways to choose i vertices to overflow.

An inclusion-exclusion argument then gives us

$$\sum_{i \geq 0} (-1)^i \binom{n}{i} \binom{n+k-3i-1}{n-1}$$

possible ways to arrange the armies without any overflowing vertices. Intuitively, it is obvious that the number of table entries where each vertex has a capacity of 2 is smaller than if there is no capacity constraint.

The number of possible table entries that need to be processed for the other time steps stays the same, so theoretically, this improvement does not amount to much. However, since the number of entries that are explored in the second time step depends on the number of entries that returned `true` in the first, the number of entries that are explored in the third time step depends on the number of entries that returned `true` in the second time step and so forth, this will reduce the required time in many practical scenarios.

Part III

Conclusion

The third and final part of the thesis will summarize the results we have obtained in the second part and reflect upon them. It will provide a table of required running times for the problems we have covered and contemplate possible ways to improve them. In addition, we will provide an overview for future research into the problems we did not solve and suggestions on how one might attempt to solve them.

11 Obtained Results

In this computing science master's thesis, we have introduced the notions of domination rules and temporal domination requirements. By combining domination rules and temporal domination requirements, we are able to define a myriad of temporal domination problems. By combining the three static domination problems Dominating Set, Roman Domination and Weak Roman Domination with the six temporal domination requirements Temporary Domination, Permanent Domination, Periodic Domination, Evolving Domination, k-Fold Domination and Marching Domination, we defined 18 temporal domination problems. In addition, we defined a natural way to extend the Defense-Like Domination problem to temporal graphs.

For all of these temporal domination problems except the Weak Roman Domination variants of the permanent, periodic and k-fold problems and for Temporal Defense-Like Domination, we have provided exact algorithms. Of the algorithms provided, those for the permanent, periodic and k-fold variants are parameterized using the treewidth ℓ as a parameter. Furthermore, all algorithms other than those for the temporary variants use τ as a parameter as well as the number of vertices n. The algorithms for all periodic variants also use the period θ as a parameter. We also provided algorithms for the decision problems corresponding to the marching variants in addition to the minimization problems. For the algorithms for the decision problems, the number of armies k is also used as a parameter. The obtained results are summarized in the following table.

Problem	**Time Complexity**	**Space Complexity**
Temporary Dominating Set	$\mathcal{O}(1.4689^n)$ using [23]	$\mathcal{O}(1.4689^n)$
Temporary Roman Domination	$\mathcal{O}(1.5014^n)$ using [28]	exponential
Temporary Weak Roman Domination	$\mathcal{O}^*(2^n)$ using [10]	exponential
Permanent Dominating Set	$\mathcal{O}\left(n^3(2^\tau+1)^{\ell+1}\right)$	$\mathcal{O}\left(n\,(2^\tau+1)^{\ell+1}\right)$
Permanent Roman Domination	$\mathcal{O}\left(n^3(2^\tau+1)^{\ell+1}\right)$	$\mathcal{O}\left(n\,(2^\tau+1)^{\ell+1}\right)$
Periodic Dominating Set	$\mathcal{O}\left(n^3\left(2^{\tau+1-\theta}+1\right)^{\ell+1}\right)$	$\mathcal{O}\left(n\left(2^{\tau+1-\theta}+1\right)^{\ell+1}\right)$
Periodic Roman Domination	$\mathcal{O}\left(n^3\left(2^{\tau+1-\theta}+1\right)^{\ell+1}\right)$	$\mathcal{O}\left(n\left(2^{\tau+1-\theta}+1\right)^{\ell+1}\right)$
Evolving Dominating Set	$\mathcal{O}(\tau 1.4689^n)$ using [23]	$\mathcal{O}(\tau 1.4689^n)$
Evolving Roman Domination	$\mathcal{O}(\tau 1.5014^n)$ using [28]	exponential
Evolving Weak Roman Domination	$\mathcal{O}^*(\tau 2^n)$ using [10]	exponential
k-Fold Dominating Set	$\mathcal{O}\left(n^3\,(2^\tau+1)^{\ell+1}\right)$	$\mathcal{O}\left(n\,(2^\tau+1)^{\ell+1}\right)$
k-Fold Roman Domination	$\mathcal{O}\left(n^3\,(2^\tau+1)^{\ell+1}\right)$	$\mathcal{O}\left(n\,(2^\tau+1)^{\ell+1}\right)$
k-Marching Dominating Set	$\mathcal{O}\left(\binom{n+k-1}{k} n d^{k+1}\tau\right)$	$\mathcal{O}\left(\binom{n+k-1}{k}\tau\right)$
k-Marching Roman Domination	$\mathcal{O}\left(\binom{n+k-1}{k} n d^{k+1}\tau\right)$	$\mathcal{O}\left(\binom{n+k-1}{k}\tau\right)$
k-Marching Weak Roman Domination	$\mathcal{O}\left(\binom{n+k-1}{k} n d^{k+3}\tau\right)$	$\mathcal{O}\left(\binom{n+k-1}{k}\tau\right)$
Marching Dominating Set	$\mathcal{O}\left(\binom{2n-1}{n} n d^{n+1}\tau\right)$	$\mathcal{O}\left(\binom{n+n-1}{n}\tau\right)$
Marching Roman Domination	$\mathcal{O}\left(\binom{2n-1}{n} n d^{n+1}\tau\right)$	$\mathcal{O}\left(\binom{n+n-1}{n}\tau\right)$
Marching Weak Roman Domination	$\mathcal{O}\left(\binom{2n-1}{n} n d^{n+3}\tau\right)$	$\mathcal{O}\left(\binom{n+n-1}{n}\tau\right)$

Here the parameters are as follows:

n : The number of vertices in a graph.

τ : The number of time steps in a temporal graph.

ℓ : The width of some nice tree decomposition of the underlying graph.

θ : The period of a periodic domination problem.

k : The number of armies in a marching domination decision problem.

d : The maximum degree over all vertex appearances.

12 Future Work

In this section, a number of research directions are suggested as topics for future research. Some of the problems introduced in this thesis remain unsolved, or can be improved. Algorithms are still required for the Weak Roman Domination variants of the permanent, periodic and k-fold domination requirements, and for and Temporal Defense-Like Domination. That last problem can also be considered for specific parameter assignments.

The Weak Roman Domination variants of the aforementioned domination requirements may be solved by using a dynamic programming approach using colors, like we did for the permanent variants of Dominating Set and Roman Domination. Veldhuizen [32] provides a way to solve Weak Roman Domination using such techniques. These techniques can likely be extended to the temporal setting in a similar fashion as we did with the permanent and k-fold variants of Dominating Set and Roman Domination, which, in turn, can be used to solve the periodic variant as well.

In addition, the methods used in this thesis to define temporal domination problems can be extended. More domination rules can be defined by using other static domination problems. The k-Tuple Domination problem mentioned in Section 4.6 requires each vertex to be dominated at least k times. The Total Domination problem asks for a vertex set S such that each vertex in the graph is adjacent to a vertex in S, in other words, vertices cannot dominate themselves. The Secure Domination Problem is identical to Weak Roman Domination, with the exception that we can only use the labels 0 and 1.

All of these static domination problems can be turned into new domination rules and we can apply a temporal domination requirement to this. For example, the permanent variant of the k-Tuple Domination problem would ask for each vertex to be dominated at least k times on every time step.

Finally, more temporal domination requirements can be designed. For example, temporal domination requirements can be considered that require each vertex to be dominated in *exactly* k time steps instead of *at least* k time steps. Other variants of Marching Domination can also be considered. One example would be a variant of Marching Domination where the number of moves needs to be minimized after minimizing the number of armies. We could also add parameters to the marching domination requirement, allowing us to limit the number of armies that can be moved, or increase the number of moves each army can make in a time step, like in Defense-Like Domination.

References

[1] Eleni C. Akrida, George B. Mertzios, Paul G. Spirakis, and Viktor Zamaraev. Temporal vertex cover with a sliding time window. *Journal of Computer and System Sciences*, 107:108–123, 2020.

[2] Jochen Alber, Hans L. Bodlaender, Henning Fernau, Ton Kloks, and Rolf Niedermeier. Fixed parameter algorithms for dominating set and related problems on planar graphs. *Algorithmica*, 33:461–493, 2002.

[3] Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Fourier meets Möbius: Fast subset convolution. In *Proceedings of the Thirty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '07, page 67–74. Association for Computing Machinery, 2007.

[4] Robert C. Brigham and Ronald D. Dutton. Factor domination in graphs. *Discrete Mathematics*, 86:127–136, 1990.

[5] Robert C. Brighanm and Julie R. Carrington. Global domination. In *Domination in Graphs: Advanced Topics*, pages 301–320. 1998.

[6] Yair Caro and Michael A. Henning. Simultaneous domination in graphs. *Graphs and Combinatorics*, 30:1399–1416, 2014.

[7] Yair Caro and Raphael Yuster. Dominating a family of graphs with small connected subgraphs. *Combinatorics, Probability and Computing*, 9:309–313, 2000.

[8] Arnaud Casteigts and Paola Flocchini. Deterministic Algorithms in Dynamic Networks: Problems, Analysis, and Algorithmic Tools, 2013. <hal-00865764 >.

[9] Arnaud Casteigts, Bernard Mans, and Luke Mathieson. On the feasibility of maintenance algorithms in dynamic graphs. *CoRR*, abs/1107.2722, 2011.

[10] Mathieu Chapelle, Manfred Cochefert, Jean-Françcois Couturier, Dieter Kratsch, Romain Letourneur, Mathieu Liedloff, and Anthony Perez. Exact algorithms for weak Roman domination. *Discrete Applied Mathematics*, 248:79–92, 2018.

[11] Ernie J. Cockayne, Paul A. Dreyer, Sandra M. Hedetniemi, and Stephen T. Hedetniemi. Roman domination in graphs. *Discrete Mathematics*, 278:11–22, 2004.

[12] Peter Dankelmann, Michael Henning, Wayne Goddard, and Renu Laskar. Simultaneous graph parameters: Factor domination and factor total domination. *Discrete Mathematics*, 306:2229–2233, 2006.

[13] Peter Dankelmann and Renu Laskar. Factor domination and minimum degree. *Discrete Mathematics*, 262:113–119, 2003.

[14] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity.* Springer-Verlag, 1999.

[15] Henning Fernau. Roman domination: a parameterized perspective. *International Journal of Computer Mathematics*, 85:25–38, 2008.

[16] Till Fluschnik, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. As time goes by: Reflections on treewidth for temporal graphs. In *Treewidth, Kernels, and Algorithms. Essays Dedicated to Hans L. Bodlaender on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, page 49–77. Springer, 2020.

[17] Fedor V. Fomin and Dimitrios M. Thilikos. Dominating sets in planar graphs: Branch-width and exponential speed-up. *SIAM Journal on Computing*, 36:281–309, 2006.

[18] Frank Harary and Teresa Haynes. Double domination in graphs. *Ars Comb.*, 55, 2000.

[19] Teresa W. Haynes, Stephen Hedetniemi, and Peter Slater. *Fundamentals of Domination in Graphs*, volume 208 of *Pure and Applied Mathematics.* Dekker, 1998.

[20] Michael A. Henning and Stephen T. Hedetniemi. Defending the roman empire: A new strategy. *Discrete Mathematics*, 266:239–251, 2003.

[21] Michael A. Henning and Adel P. Kazemi. k-Tuple total domination in graphs. *Discrete Applied Mathematics*, 158:1006–1011, 2010.

[22] Niklas Hjuler, Giuseppe F. Italiano, Nikos Parotsidis, and David Saulpic. Dominating sets and connected dominating sets in dynamic graphs. *CoRR*, abs/1901.09877, 2019.

[23] Yoichi Iwata. A faster algorithm for dominating set analyzed by the potential method. In *Proceedings of the 6th International Conference on Parameterized and Exact Computation*, page 41–54. Springer-Verlag, 2011.

[24] Ton Kloks. *Treewidth, Computations and Approximations*, volume 842 of *Lecture Notes in Computer Science*. Springer, 1994.

[25] Hendrik Molter. Parameterized algorithmics for temporal graph problems. *Parameterized Complexity Newsletter*, 16:4–5, 2020.

[26] Tim Nieberg and Johann L. Hurink. A PTAS for the Minimum Dominating Set Problem in Unit Disk Graphs. In *Approximation and Online Algorithms*, volume 3879 of *Lecture Notes in Computer Science*, pages 296–306. Springer, 2005.

[27] Fred Roberts and Barry Tesman. *Applied Combinatorics (2. ed.)*. CRC Press, 2005.

[28] Zheng Shi and Khee Meng Koh. Counting the number of minimum Roman dominating functions of a graph. *CoRR*, abs/1403.1019, 2014.

[29] Ian Stewart. Defend the Roman Empire! *Scientific American*, 281:136–138, 1999.

[30] Johan M. M. van Rooij, Hans L. Bodlaender, and Peter Rossmanith. Dynamic programming on tree decompositions using generalised fast subset convolution. In Amos Fiat and Peter Sanders, editors, *Algorithms - ESA 2009*, volume 5757 of *Lecture Notes in Computer Science*, pages 566–577. Springer, 2009.

[31] Johan M.M. van Rooij. Exact Exponential-Time Algorithms for Domination Problems in Graphs. *Phd thesis, Utrecht University*, 2011.

[32] Mats Veldhuizen. Exact Exponential-Time and Treewidth-Based for Defense-like Domination Problems. Master's thesis, Utrecht University, 2020.