Technische Universität Berlin
Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Algorithmics and Computational Complexity (AKT)

Leveraging Graph Structure to Untangle Temporal Networks Efficiently

Carsten Schubert

Thesis submitted in fulfillment of the requirements for the degree
“Master of Science” (M. Sc.) in the field of Computer Science

Oktober 2023

Supervisor and first reviewer:	Prof. Dr. Mathias Weller
Second reviewer:	Prof. Dr. Martin Skutella
Co-Supervisors:	Pascal Kunz and Philipp Zschoche

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der angeführten Quellen und Hilfsmittel angefertigt habe.

Die selbstständige und eigenhändige Ausfertigung versichert an Eides statt

Berlin, den	____________________	____________________________
	Datum	Unterschrift

Abstract

In this work, we study the parameterized computational complexity of the NETWORK UNTANGLING problem, which has recently emerged from applications with a need to summarize large temporal networks. This temporal version of the well-known VERTEX COVER problem asks whether all time-edges in a given temporal graph can be covered, if each vertex is only allowed to cover incident edges for a limited number of time step intervals. It thereby helps with identifying entities that are of special importance for several short time spans in the network. We investigate two variants of this problem, which differ in the way these time spans are restricted: One of them requires that each individual vertex interval has a short length, while the other demands that the total sum of all these intervals is small.

Both variants are NP-hard, which is why efficient parameterized algorithms can be an especially powerful tool to solve various restricted NETWORK UNTANGLING instances. Froese, Kunz, and Zschoche (2022) have already initiated the search for fixed-parameter-tractable cases of both problem variants. However, they only studied specific input parameters of NETWORK UNTANGLING in combination with the number of vertices n, not analyzing any other graph structural parameters. Hence, in their concluding remarks, they emphasized the need for further research on the parameterized complexity regarding graph parameters smaller than n.

Following that suggestion, we now almost completely clarify the problems' parameterized complexity concerning the graph parameters treewidth (**tw**), tree partition width (**tpw**), edge-treewidth (**etw**), feedback vertex set number (**fvs**) and vertex cover number (**vc**), each combined with the problem-specific parameters k_∞, ℓ and τ. Most notably, we provide novel FPT-algorithms for both problem variants w.r.t. $\mathbf{tw} + \tau$ and w.r.t. $\mathbf{tpw} + k_\infty + \ell$. We also present various new parameterized hardness results and prove a previously unknown parameter relationship between **tpw** and **etw**.

Deutsche Zusammenfassung

In dieser Arbeit untersuchen wir die parameterisierte Komplexität des Problems der sogenannten „NETZWERK-ENTWIRRUNG“ (englisch „NETWORK UNTANGLING“). Dieses Problem, eine temporale Variante des allgemein bekannten Knotenüberdeckungsproblems, wurde kürzlich aus diversen Anwendungen hergeleitet, die das Ziel haben, die Informationen großer temporaler Netzwerke zusammenzufassen. Konkret stellt das Problem die Frage, ob alle Zeit-Kanten eines gegebenen temporalen Graphen überdeckt werden können, falls jeder Netzwerkknoten seine angrenzenden Kanten nur für eine beschränkte Zahl an Zeitintervallen überdecken kann. Somit hilft das Problem dabei, für kurze Zeitspannen besonders wichtige Entitäten im Netzwerk zu identifizieren. Wir betrachten zwei Versionen dieses Problems, welche sich in der Art der geforderten Beschränkung dieser Zeitspannen unterscheiden: In einer davon verlangen wir, dass jedes individuelle Zeitintervall kurz ist. In der anderen Version geht es darum, die Gesamtsumme aller Intervalllängen zu minimieren.

Beide Problemfälle sind NP-schwer, weshalb effiziente parameterisierte Algorithmen eine besonders wirkungsvolle Methode darstellen können, um dennoch bestimmte Instanzen dieser Probleme zu lösen. Froese, Kunz und Zschoche (2022) haben bereits einige FPT-Algorithmen für beide Varianten formulieren können. Allerdings haben sie dabei nur bestimmte problemspezifische Parameter in Kombination mit der Anzahl der Knoten des Netzwerks n untersucht und sich keine kleineren strukturellen Graphparameter als n angesehen. Entsprechend haben sie auch in ihren Schlussworten darauf hingewiesen, dass zukünftige Forschung sich der Frage widmen solle, ob parameterisierte Algorithmen bezüglich kleinerer Graphparameter als n zur NETZWERK-ENTWIRRUNG existieren.

Wir nehmen uns nun diesem Vorschlag an und erkunden die parameterisierte Landschaft der beiden Problemversionen fast vollständig hinsichtlich der Graphparameter Baumweite (**tw**), Baumpartitionsweite (**tpw**), Kanten-Baumweite (**etw**), kreiskritische Knotenzahl (**fvs**, englisch „Feedback Vertex Number“) und Knotenüberdeckungszahl, in Kombination mit den problemspezifischen Parametern k_∞, ℓ und τ. Besonders ist dabei hervorzuheben, dass wir neue FPT-algorithmen für die kombinierten Parameter $\mathbf{tw} + \tau$ sowie $\mathbf{tpw} + k_\infty + \ell$ (jeweils für beide Versionen des Problems) präsentieren können. Weiterhin stellen wir mehrere vorher unbekannte parameterisierte Härtefälle vor und zeigen eine ebenfalls bislang unbekannte Parameterbeziehung zwischen **tpw** und **etw** auf.

Contents

1 Introduction

The analysis of temporally changing networks has emerged as a major trend in the domain of data analysis in recent years. Many real-life phenomena which develop over time can be modeled using temporal graphs, combining the already large body of knowledge on conventional networks with a time-dependant aspect (Holme and Saramäki 2012; Wang et al. 2019). However, summarizing large amounts of temporal network data (i.e. extracting concentrated information) is often challenging—both in the sense of finding a representative network model for a particular application and in the sense of algorithmically computing optimal solutions in a model (Holme 2015; Fluschnik et al. 2020b; Molter, Renken, and Zschoche 2021).

A recently introduced notion of extracting relevant data from temporal graphs is suggested by the NETWORK UNTANGLING problem: Herein, we search for timelines consisting of different network entities at different time intervals which together relate to every recorded interaction. The aim is thus to explain all interactions while needing only a relatively small number of vertices for a small number of consecutive time steps each.

Applications involve e.g. identifying users in a social network who spread certain sensitive information at different times, or finding positions from which an ever-changing environment can be fully monitored if each position is usable for a short amount of time only. Futhermore, Rozenshtein, Tatti, and Gionis (2021) describe the use case of pinpointing important past events from a given chronology of online news data. Computed solutions in such a scenario could additionally be of use as labeled training data to adapt supervised learning algorithms for similar settings that may occur in the future.

NETWORK UNTANGLING is proved to be NP-hard, even in several restricted settings (Froese, Kunz, and Zschoche 2022). As a consequence, there are no algorithms for efficiently computing solutions in general (unless P $=$ NP). Still, some algorithmic strategies have been proposed, for instance using heuristics (Rozenshtein, Tatti, and Gionis 2021) or parameterized methods (Froese, Kunz, and Zschoche 2022; Dondi and Lafond 2023). We focus on the latter and explore graph structural properties to advance the previously existing research.

In particular, the only graph parameter previously considered for parameterization of all NETWORK UNTANGLING variants was the number of vertices in the input (Froese, Kunz, and Zschoche 2022), which is usually rather large. Accordingly, in their conclusion, the authors of the respective work pointed out that smaller parameters like the underlying graph's treewidth or its vertex cover number should be considered for potential FPT-algorithms by further research. We follow this train of thought, thereby finding new special cases where we obtain efficient computation and new parameterized hardness results alike (see Table 1.1 for details). We especially exploit certain “tree-like” input structures, combined with other small input parameters, to present efficient algorithms.

1.1 Problem Definitions

Network Untangling was introduced by Rozenshtein, Tatti, and Gionis (2021) and subsequently further analyzed by Froese, Kunz, and Zschoche (2022), Dondi (2022, 2023), Dondi and Lafond (2023), and Dondi and Popa (2023). In this problem, vertices of a given temporal graph can be *activated* at particular time steps and stay *active* for a restricted amount of time. Intuitively, Network Untangling then asks whether all edges in all layers of the temporal graph can be covered by active vertices if each vertex is activated only a limited number of times. In a layer, an edge is *covered* if it is incident to an active vertex. In this sense, the problem can be seen as a temporal version of the well-known Vertex Cover problem, since we are looking for a vertex cover in every layer. However, note that these vertex covers do not need to be of minimum size. Instead, we require the total number of activations for each vertex to be restricted.

In order to formally define the problem, we first introduce the notion of *k-activity timelines*.

Definition 1.1: k-Activity Timeline

Given a temporal graph $\mathcal{G} = (V, E_1, \dots, E_\tau)$ and a vector k with an entry $k_v \in \mathbb{N}$ for each $v \in V$, a k-activity timeline is a set $\mathcal{C} \in V \times [\tau] \times [\tau]$ with

- $a \leq b$ for each $(v, a, b) \in \mathcal{C}$,
- $|\{(v, a, b) \in \mathcal{C}\}| \leq k_v$ for each $v \in V$, and
- if $\{v, w\} \in \mathcal{G}_t$ for any layer $\mathcal{G}_t$ of $\mathcal{G}$, then $(v, a, b) \in \mathcal{C}$ or $(w, a, b) \in \mathcal{C}$ for some $a \leq t \leq b$.

Each $(v, a, b) \in \mathcal{C}$ is called an *interval* of vertex v from time step a to b with *length* $(b - a)$. The vertex v is *activated* at time step a and is *active* at each time step $t \in [a, b]$ if $(v, a, b) \in \mathcal{C}$. We call any $(v, t) \in V \times [\tau]$ a *vertex occurence*. The last condition in Definition 1.1 is thus also referred to as *covering* all temporal edges using vertex occurences (i.e. vertices at time steps) which belong to intervals of $\mathcal{C}$.

We study two variants of Network Untangling, which differ in the way that timeline intervals are restricted:

Problem Definition 1.2: (Non-Uniform) MinTimeline$_\infty$

Input: A temporal graph $\mathcal{G} = (V, E_1, \dots, E_\tau)$, a vector $k \in \mathbb{N}^{|V|}$, and a number $\ell \in [0, \tau - 1]$.
Question: Is there a k-activity timeline $\mathcal{C}$ for $\mathcal{G}$ and k, such that $b - a \leq \ell$ for all $(v, a, b) \in \mathcal{C}$?

Problem Definition 1.3: (Non-Uniform) MinTimeline$_+$

Input: A temporal graph $\mathcal{G} = (V, E_1, \dots, E_\tau)$, a vector $k \in \mathbb{N}^{|V|}$, and a number $\ell \in \mathbb{N}$.
Question: Is there a k-activity timeline $\mathcal{C}$ for $\mathcal{G}$ and k, such that $\sum\limits_{(v,a,b)\in\mathcal{C}} (b - a) \leq \ell$?

Intuitively, in MinTimeline$_\infty$ (the "maximum variant") each interval has a fixed length of ℓ. In MinTimeline$_+$ (the "sum variant"), all interval lengths may sum up to at most ℓ. Note that both variants are identical as long as $\ell = 0$.

1.2 Our contributions

Table 1.1 summarizes the contributions of this work and references the corresponding theorems. Since many results are derived via parameter relationships (of which we even discovered a previously unknown one), we provide an overview of all graph parameters investigated in this work, as well as their relationships, in Figure 1.1. If not stated otherwise, our conception of both Network Untangling variants is non-uniform, meaning that we allow each vertex to have a separate number of activation intervals. This slightly differs from previous problem definitions, e.g. by Rozenshtein, Tatti, and Gionis (2021). However, all of our results can be modified to work with those uniform problem formulations as well (see Chapter 4).

We especially highlight that we formulated FPT-algorithms for both problem variants concerning the combined parameter **tw** $+ \tau$ (Algorithms 3.1 and 3.2) as well as **tpw** $+ k + \ell$ (Algorithms 3.3 and 3.4), but showed that no such algorithm exists regarding the parameter **tw** $+ k + \ell$ (unless W[2] $=$ FPT).

Graph Parameter Landscape

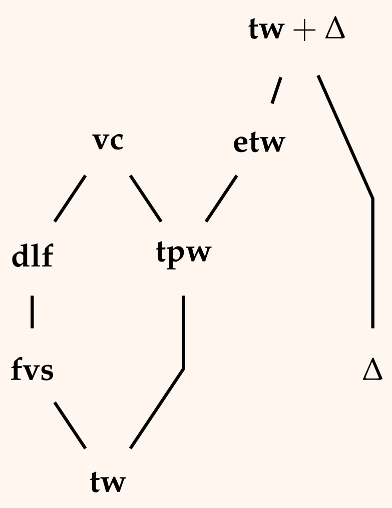

Figure 1.1: Outline of all mentioned graph parameters and their relationships in the form of a Hasse Diagram. This means parameter A upper-bounds B if there is a path from A to B which only uses downward-going lines. E.g. the edge-treewidth (**etw**) upper-bounds the treewidth (**tw**) of any simple graph, but not vice-versa. If two parameters are not shown to have a relationship where one upper-bounds the other, then they are known to be parametrically incomparable—like the tree partition width (**tpw**) and the feedback vertex set number (**fvs**). Other depicted parameters include the distance to a linear forest (**dlf**), the vertex cover number (**vc**) and the maximum degree (Δ). The **etw** parameter was introduced by Magne et al. (2023), definitions of the other depicted graph parameters can be found in Section 1.4.2. The relationship between **etw** and **tpw** is proved in Theorem 3.21.

Parameterized Complexity of NON-UNIFORM NETWORK UNTANGLING			
+	τ	$k_\infty + \ell$	ℓ
Δ	paraNP-hard (on bipartite graphs) [Thm. 2.7]	paraNP-hard (on bipartite graphs) [Thm. 2.7]	paraNP-hard (on caterpillar trees) [Thm. 2.4]
dlf	FPT [Thm. 3.1,Thm. 3.4]	XP, W[2]-hard (with **dlf** $= 1$) [Thm. 3.1,Thm. 3.4, Thm. 3.8]	paraNP-hard (on stars) [Thm. 2.1]
fvs	FPT [Thm. 3.1,Thm. 3.4]	XP, W[2]-hard (with **fvs** $= 1$) [Thm. 3.1,Thm. 3.4, Thm. 3.8]	paraNP-hard (on stars) [Thm. 2.1]
tw	FPT [Thm. 3.1,Thm. 3.4]	XP, W[2]-hard (with **tw** $= 2$) [Thm. 3.1,Thm. 3.4, Thm. 3.8]	paraNP-hard (on stars) [Thm. 2.1]
tpw	FPT [Thm. 3.1,Thm. 3.4]	FPT [Thm. 3.12,Thm. 3.17]	paraNP-hard (on stars) [Thm. 2.1]
vc	FPT [Thm. 3.1,Thm. 3.4]	FPT [Thm. 3.12,Thm. 3.17]	paraNP-hard (on stars) [Thm. 2.1]
etw	FPT [Thm. 3.1,Thm. 3.4]	FPT [Cor. 3.23]	paraNP-hard (on caterpillar trees) [Thm. 2.4]
tw $+ \Delta$	FPT [Thm. 3.1,Thm. 3.4]	FPT [Thm. 3.12,Thm. 3.17]	paraNP-hard (on caterpillar trees) [Thm. 2.4]

Table 1.1: Summary of our parameterization results regarding MINTIMELINE$_\infty$ and MINTIMELINE$_+$. Each row represents a graph parameter for the underlying graph of the input and each column represents an additional input parameter. Consequently, a table cell displays the classification of NETWORK UNTANGLING w.r.t. a combination of the corresponding two parameters. The graph parameter abbreviations and their relationships (using which many shown results are derived) are described in Figure 1.1. The parameters on the columns are the number of temporal layers τ, the maximal number of activations per vertex k_∞ and the interval length bound ℓ (see Section 1.1 for detailed explanations). All results apply to both studied variants of NETWORK UNTANGLING similarly. Every hardness result holds even if $\ell = 0$.

1.3 Related Work

As already stated, UNIFORM MINTIMELINE$_\infty$ and UNIFORM MINTIMELINE$_+$ as well as their general term "NETWORK UNTANGLING" were introduced by Rozenshtein, Tatti, and Gionis (2021), who showed that both problems are NP-hard and APX-hard (in our formulation with unbounded k). They also provided and tested some heuristic approaches to solve these problems. Subsequently, Froese, Kunz, and Zschoche (2022) initiated the exploration of parameterized algorithms for both problem variants, for example by discovering that both are in FPT w.r.t. $n + k$ (n being the number of vertices). Moreover, they derived the non-uniform variants of NETWORK UNTANGLING, which we use by default. Thus, our paper builds on their foundation by refining the parameterization and exploring novel algorithmic approaches. For instance, we strengthen their result of paraNP-hardness w.r.t. $\tau + k + \ell$ by showing that it also holds on bipartite graphs. Furthermore, we present algorithms to solve both variants in FPT-time w.r.t. $\mathbf{tpw} + k_\infty + \ell$, which is more fine-tuned in comparison to their $n + k$ algorithm at the expense of including ℓ in the time bound.

The special variant of UNIFORM MINTIMELINE$_+$ where $k = 1$ has been extensively researched by Dondi (2022, 2023), Dondi and Lafond (2023), and Dondi and Popa (2023), who proved it to be fixed-parameter-tractable w.r.t. ℓ, for instance. Another problem relatively similar to NETWORK UNTANGLING with $\ell = 0$ is TEMPORAL VERTEX COVER as introduced by Akrida et al. (2020). The main difference to our scenario is that in their setting each edge of the underlying graph needs to be covered in only one layer, whereas we require it to be covered in every layer where it is present. As a consequence, we also drop their condition to find minimum vertex covers. Of course, if $\ell \neq 0$, our problems also add intervals to the vertex appearances. In contrast, they use a notion of sliding windows to fit vertex covers into the flow of time instead. In a broader view, the literature also contains very different variants of vertex covering on temporal graphs (Fluschnik et al. 2022) and dynamic graphs (Iwata and Oka 2014; Alman, Mnich, and Williams 2020).

1.4 Preliminaries

We denote by $\mathbb{N}$ and $\mathbb{N}^+$ the natural numbers including and excluding zero, respectively. The short notation $[n]$ for any $n \in \mathbb{N}$ describes the set of natural numbers from 1 to n, i.e. $[n] = \{x \in \mathbb{N} \mid 1 \leq x \leq n\}$. Similarly, we declare $[n, m]$ for any $n, m \in \mathbb{N}$ as the set $\{x \in \mathbb{N} \mid n \leq x \leq m\}$. For any set S, we denote its power set by $\mathcal{P}(S)$. For any vector k, k_i refers to its i-th entry and $|k|$ is its number of entries. Sometimes we will use other identifiers than natural numbers for i, in these cases the order of entries in k is not relevant (just the mapping from i to k_i is). k_∞ represents the maximum norm of a vector k, i.e. $k_\infty = \max\limits_{i \in [|k|]} |k_i|$.

1.4.1 Basic Graph Theory

A (simple) graph $G = (V, E)$ is a structure which contains a finite non-empty set V of *vertices* (or *nodes*) and a finite set $E \subseteq \{\{u, w\} \mid \{u, w\} \subseteq V, u \neq w\}$ of *edges*. The vertices u and w are called *endpoints* of the edge $\{u, w\}$ and the edge lies *between* (or *is incident to*) its two endpoints. We say that a vertex v or an edge $e = \{a, b\}$ is (included or contained) in G if $v \in V$ or $e \in E$, respectively. The former can also be denoted by $v \in G$ and the latter can also be denoted by $\{a, b\} \in G$ if clear in context. We define the size of G, denoted by $|G|$, as the total number of vertices and edges in G. A graph $G' = (V', E')$ is a *subgraph* of another graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$. The *induced subgraph* of a vertex set $S \subseteq V$ for a graph $G = (V, E)$, denoted by $G[S]$, is the graph $G' = (S, E')$, where $E' = \{\{a, b\} \in G \mid \{a, b\} \subseteq S\}$. Further, $G \setminus S$ describes the graph obtained by deleting vertices S, i.e. $G \setminus S = G[V \setminus S]$. When clear from context, we may also write $G \setminus v$ instead of $G \setminus \{v\}$ for a vertex v.

The set of *neighbors* $N(v)$ of a vertex v in a graph $G = (V, E)$ is defined as $N(v) = \{w \in V : \{v, w\} \in E\}$. Two vertices are *adjacent* if they are neigbors. The *closed neighborhood* of v, meaning the neighbors of v together with v itself will be denoted by $N[v]$, i.e. $N[v] = N(v) \cup \{v\}$. If it is otherwise unclear which graph we are referring to, we will specify this graph explicitly, like $N_G(v)$ or $N_G[v]$. The *degree* of a vertex v in a graph $G = (V, E)$ refers to the number of its neigbors and is abbreviated by $\deg_G(v)$. Again, if the graph is clear from context we will just use $\deg(v)$.

A *path* P_n is a graph with $n \in \mathbb{N}^+$ vertices which can be relabeled and listed as the sequence $(v_1, \ldots, v_n)$, such that $\{\{v_i, v_{i+1}\} \mid i \in [n-1]\}$ is the edge set of P_n. We say that this path *starts* in the vertex relabeled as v_1 and *ends* in the vertex relabeled as v_n. A graph G is *connected*, if for each two distinct vertices a and b included in G there is a path subgraph of G starting in a and ending in b. If there is exactly one such path subgraph from each vertex a to each other vertex b in G, then G is called a *tree*. When picking some vertex r in a tree T as *root (node)*, the structure becomes a *rooted tree* (T, r). In such a rooted tree each vertex v with $v \neq r$ has a unique *parent* w, which is the next vertex after v on the path starting in v and ending in r. If w is the parent of v, then v is w's *child*. This means that each vertex w in a rooted tree has a unique set of *children*. If this set is empty, then w is called a *leaf (node)*. The set of *ancestors* of v contains all vertices in the path starting in v's parent and ending in the root node r (if $v = r$, then v has no ancestors). Similarly, the set of *descendants* of v consists of all vertices which are in any path starting in a child of v and ending in some leaf node. The *subtree* of vertex v in a rooted tree is the induced subgraph of v and its descendants, usually viewed as rooted in v.

A (connected) *component* C of G is an inclusion-maximal connected subgraph of G. Note that if G is connected, it contains only one component. A *forest* is a graph where each of its components is a tree and a *linear forest* is a graph where each of its components is a path. A vertex $s \in G$ is called *cut vertex*, if the number of components of $G \setminus s$ is larger than the number of components of G. A *biconnected component* or *block* of G is an inclusion-maximal connected subgraph of G which in itself contains no cut vertices. Thus, the only vertices in G which are contained in multiple blocks are the cut vertices of G.

A connected graph in which the vertices of degree higher than one induce a path subgraph is called a *caterpillar tree*. A *star* is a connected graph where there is one special *center* vertex which forms an endpoint of each edge in the graph. Thus, each other vertex in the star has a degree of one. We usually denote the star with $n + 1$ vertices by S_n for each $n \in \mathbb{N}$. The sets $X_1, \dots, X_n$ form a *partition* of a set X if each of these X_i is a non-empty subset of X and each $x \in X$ is contained in exactly one of these X_i. A graph G is called *bipartite* if its vertex set can be partitioned into two sets L and R such that each edge in G has one endpoint in L and the other endpoint in R. Usually, L is called the left side and R is called the right side of the graph. A *complete bipartite graph* is a graph $G = (V, E)$ with such a partition (L, R) where $E = \{\{a, b\} \mid a \in L, b \in R\}$. A complete bipartite graph with $n \in \mathbb{N}^+$ vertices on the left and $m \in \mathbb{N}^+$ vertices on the right side is denoted by $K_{n,m}$.

1.4.2 Graph Parameters

Next we want to define some parameters which are special for graphs. A *graph parameter* is a function from a graph to a natural number (including zero).

The *maximum degree* $\Delta(G)$ describes the highest degree among vertices in G, i.e. $\Delta(G) = \max\limits_{v \in G}(\deg_G(v))$. A *vertex cover* is a set of vertices $S \subseteq V$ of a graph $G = (V, E)$ such that each edge in G has an endpoint in S, i.e. $e \cap S \neq \emptyset$ for each $e \in E$. The *vertex cover number* $\mathbf{vc}(G)$ describes the lowest natural number x for which there exists a vertex cover S of graph G with $|S| = x$. A *feedback vertex set* $S \subseteq V$ in a graph $G = (V, E)$ is a set of vertices such that $G \setminus S$ is a forest. The *feedback vertex set number* $\mathbf{fvs}(G)$ is the lowest number x for which there exists a feedback vertex set of size x in G. Similarly, the *distance to linear forest* $\mathbf{dlf}(G)$ is the lowest number x for which there exists a set of vertices $S \subseteq V$ in $G = (V, E)$ with $|S| = x$ and $G \setminus S$ is a linear forest. A *k-coloring* for $k \in \mathbb{N}^+$ of a graph $G = (V, E)$ is a function from V to $[k]$. Informally speaking, this function assigns a color $c \in [k]$ to each vertex in G. A k-coloring is *proper* (or *valid*) if for each $\{a, b\} \in E$ the vertices a and b are assigned different colors.

A *tree decomposition* of a graph $G = (V, E)$ is a tree $D = (B, E')$, where

- $\bigcup\limits_{b \in B} b = V$,
- for each $\{a, b\} \in E$ there exists at least one $b \in B$ with $\{a, b\} \subseteq b$, and
- if $a \in b_1 \cap b_2$ for two distinct vertices $b_1, b_2 \in B$, then also $a \in b_3$ for each vertex $b_3 \in B$ included in the unique path from b_1 to b_2 in D.

The vertices B of a tree decomposition are also called *bags*. The *treewidth* $\mathbf{tw}(G)$

is the lowest number x, such that there is a tree decomposition D of G where each bag in D has size at most $x + 1$.

A *tree partition* of a graph $G = (V, E)$ is also a tree $D = (B, E')$ whose vertices are subsets of V and are often called bags. However, for a tree partition it holds that

- each $v \in V$ is contained in exactly one $b \in B$, and
- for each $\{a, b\} \in E$ there exists either a bag $b \in B$ with $\{a, b\} \subseteq b$ or two bags b_1 and b_2 which are neigbors in D and where $a \in b_1$ and $b \in b_2$.

Accordingly, the *tree partition width* $\mathbf{tpw}(G)$ is the lowest number x, such that there is a tree partition D of G where each bag in D has size at most x.

If the graph is clear from context, we will sometimes just write p instead of $p(G)$ for any graph parameter p.

1.4.3 Temporal Graphs

The concept of *temporal graphs* is a generalization of simple graphs, introducing the possibility to feature multiple edge sets. Specifically, a temporal graph is a tuple $\mathcal{G} = (V, E_1, \ldots, E_\tau)$ with $\tau \in \mathbb{N}$, where $\mathcal{G}_i = (V, E_i)$ is a simple graph for each *time step* $i \in [\tau]$. These $\mathcal{G}_i$ are called *layers* or *snapshots* of $\mathcal{G}$.

Equivalently, one can define a temporal graph by introducing *timestamps* to the edges of a simple graph, i.e. $\mathcal{G} = (V, \mathcal{E})$ with $\mathcal{E} \subseteq E \times [\tau]$, $\tau \in \mathbb{N}$ and (V, E) being a simple graph. We call each $(\{a, b\}, t) \in \mathcal{E}$ a *temporal edge* of $\mathcal{G}$ with timestamp t. We will use those equivalent definitions interchangeably and notate $v \in \mathcal{G}$ as well as $(\{a, b\}, t) \in \mathcal{G}$ for vertices and temporal edges in $\mathcal{G}$ like we did for simple graphs. We also use the basic terminology we established for simple graphs, if unambiguous.

The *underlying graph* $\mathcal{G}_\downarrow$ of a temporal graph $\mathcal{G} = (V, E_1, \ldots, E_\tau)$ is the simple graph obtained from joining all its edge sets together, i.e. $\mathcal{G}_\downarrow = (V, \bigcup\limits_{i \in [\tau]} E_i)$. In order to use our previous graph parameters on temporal graphs as well, we define temporal graph parameters by using the underlying graph as intermediate: $p(\mathcal{G}) = p(\mathcal{G}_\downarrow)$ for each temporal graph $\mathcal{G}$ if p is a parameter defined on simple graphs.

If not otherwise stated, we will assume that all temporal graphs we work with have connected underlying graphs. Further, we will usually describe simple graphs as just *graphs*, which we do not do for temporal graphs to avoid confusion.

1.4.4 Classical and Parameterized Complexity Theory

Let Σ be a finite alphabet. In classical complexity theory, a *problem* is defined as a language $L \subseteq \Sigma^*$. Given an *instance* $x \in \Sigma^*$ of L the task then arises to decide whether x is a YES-instance, i.e. $x \in L$, or a NO-instance ($x \notin L$). Two instances x, x' of problems L, L' are *equivalent* if $x \in L \Leftrightarrow x' \in L$.

In parameterized complexity theory, we study problems always with respect to some problem parameter $k \in \mathbb{N}$. A *parameterized problem* L is then a subset $L \subseteq \{(x,k) \in \Sigma^* \times \mathbb{N}\}$. Again, an instance (x,k) of L is a YES-instance if $(x,k) \in L$ and a NO-instance otherwise. We also call two instances $(x,k), (x',k')$ of parameterized problems L, L' equivalent if $(x,k) \in L \Leftrightarrow (x',k') \in L$. We call a parameterized problem L fixed-parameter tractable or in FPT if there is an algorithm deciding for each input instance (x,k) if it is a YES-instance of L in $f(k) \cdot |x|^{O(1)}$ time, where f is some computable function which only depends on k. Similarly, we say that a parameterized problem L is in XP if there is an algorithm deciding for each input instance (x,k) if it is a YES-instance of L in $|x|^{f(k)}$ time, again f being some computable function only dependent on k.

In classical complexity theory we often show NP-hardness of a problem L' by describing a polynomial-time many-one ("Karp") reduction from an NP-hard problem L to L', i.e. an algorithm that takes an input instance x of L and then generates in $O(|x|^{O(1)})$ time an instance x' of L' such that x and x' are equivalent. This equivalence property is called *correctness* of the reduction. If a problem is NP-hard, it is assumed to not be decidable in polynomial time (w.r.t. its input size). If a problem is paraNP-hard regarding some parameter, i.e. NP-hard on instances of which the respective parameter is constant, then it is assumed to not be contained in XP, analogously.

In a similar way, in parameterized complexity theory we often show W$[t]$-hardness of a parameterized problem L' for any $t \in \mathbb{N}$ by describing a *parameterized reduction* from a W$[t]$-hard problem L to L'. This is an algorithm which takes an input instance (x,k) of L and computes in $f(k)|x|^{O(1)}$ time an equivalent instance (x',k') of L' such that k' is upper-bounded by $g(k)$, where both f and g are computable functions only depending on k. A W[1]-hard problem is not fixed-parameter tractable unless W$[1] =$ FPT and each W$[t+1]$-hard problem is also W$[t]$-hard for any $t \in \mathbb{N}$.

Obviously, any meaningful parameter p can equivalently be viewed as a function $p : L \to \mathbb{N}$ which maps instances of a certain classical problem L to a natural number. $(x, p(x))$ with $x \in L$ then forms the respective parameterized problem instance. In this sense, we often use graph parameters as previously defined to study the parameterized complexity of problems involving graphs or temporal graphs. We will use these two perspectives on the term "parameter" interchangeably.

Let $p : L \to \mathbb{N}$ and $p' : L \to \mathbb{N}$ both be parameters (in the function format) for the same problem L. We say that p is *(polynomially) upper-bounded* by p' if there is a computable (polynomial) function $f : \mathbb{N} \to \mathbb{N}$ such that $p(x) \leq f(p'(x))$ for each $x \in L$. If p' (polynomially) upper-bounds p and p also (polynomially) upper-bounds p', we say that p and p' are *(polynomially) tied* or *(polynomially) equivalent*. If neither p upper-bounds p' nor p' upper-bounds p, then p and p' are *incomparable*.

2 NP-Hardness of Network Untangling on Restricted Graph Classes

This chapter is dedicated to presenting different scenarios where MINTIMELINE$_\infty$ and MINTIMELINE$_+$ are NP-hard and where consequently no efficient algorithm deciding these problems is expected to exist. Concretely, in Section 2.1 we show that both variants are NP-hard on temporal graphs with underlying tree graphs, if there are no further restrictions. In Section 2.2, we strengthen a previous NP-hardness result by extending it to hold on bipartite temporal graphs. We will later build upon all that knowledge to detect cases for which efficient algorithms exist.

2.1 NP-Hardness on Trees

Trees are one of the most fundamental graph classes. Containing no cycles by definition, they are usually regarded as rather easily approachable graphs from the algorithmic perspective and indeed many classical NP-hard graph problems are efficiently decidable or outright trivial if their input graph is a tree (e.g. INDEPENDENT SET, DOMINATING SET, or COLORING). In the setting of temporal graphs, some otherwise NP-hard problems are solvable in polynomial time on underlying trees as well (Fluschnik et al. 2020a; Fluschnik et al. 2023).

However, this is unfortunately not the case for our NETWORK UNTANGLING problems, even if $\ell = 0$. As we show in this section, both variants stay NP-hard on temporal graphs whose underlying graph is a tree. Moreover, we can restrict this condition further to either stars, or trees with a constant maximum degree.

2.1.1 Star Graphs

As already indicated, we want to prove the following:

Theorem 2.1: NP-Hardness on Star Graphs

MINTIMELINE$_+$ and MINTIMELINE$_\infty$ are both NP-hard even if the underlying graph is a star, each layer contains at most 3 edges, and $\ell = 0$.

We show Theorem 2.1 by providing a reduction from the NP-hard VERTEX COVER problem to MINTIMELINE$_\infty$ (which is identical to MINTIMELINE$_+$ as long as $\ell = 0$). The reduction closely resembles a reduction presented by Akrida et al. (2020) from the HITTING SET problem to their TEMPORAL VERTEX COVER problem. This is due to the fact that HITTING SET is a generalization of VERTEX COVER and TEMPORAL VERTEX COVER bears some similarity to MINTIMELINE$_\infty$ with $\ell = 0$ as mentioned in Section 1.3.

Problem Definition 2.2: VERTEX COVER

Input: A graph $G = (V, E)$ and an integer k.
Question: Is there a vertex cover of size at most k within G?

VERTEX COVER is NP-hard even if the maximum degree of G is 3 (Garey, Johnson, and Stockmeyer 1976), so we assume to be given a respective input graph to our reduction. We also assume its vertices are named 1 to $|V|$.

Reduction 2.3: Let $V' = E \cup \{c\}$, where c is a new vertex. Construct a temporal graph $\mathcal{G}$ with vertex set V' by creating a layer $\mathcal{G}_v$ for each $v \in V$ which contains the edges $\{\{c, e\} \mid e \in E, v \in e\}$.

For the output vector k', set $k'_c = k$ and set $k'_e = 1$ for each vertex $e \in E$. The whole output instance of MINTIMELINE$_\infty$ is now $(\mathcal{G} = (V', E_1, \ldots, E_{|V|}), k', 0)$. ♦

Using Reduction 2.3, we can immediately prove Theorem 2.1.

Proof of Theorem 2.1. It is easy to see that the temporal graph constructed by Reduction 2.3 has an underlying star graph with vertex c at its center. Also, each constructed layer has at most three edges, as $\Delta(G) = 3$. It remains to show that the reduction's input and output instances are equivalent.

($\Rightarrow$) If the input is a YES-instance of VERTEX COVER, let $S \subseteq V$ be its solution. For the solution to the output instance we activate vertex c in each layer $\mathcal{G}_v$, $v \in S$. This is possible, as $|S| \leq k = k'_c$. Observe that for each constructed vertex $e \in E$, at least one of its two incident temporal edges is now covered by c, because S is a vertex cover in the input instance. Thus, the remaining temporal edge incident to any vertex $e \in E$ can be covered by e itself with $k'_e = 1$.

($\Leftarrow$) Assume the constructed instance is YES with a solution K. Let $S' \subseteq [|V|]$ be the set of numbers of constructed layers where vertex c is activated in K. Observe that $|S| \leq k$. As each vertex $e \in E$ is adjacent to c in exactly two layers $\mathcal{G}_a$ and $\mathcal{G}_b$, K is a solution, and $k'_e = 1$, we know that $a \in S'$ or $b \in S'$. Since vertex e was constructed for the input edge $e = \{a, b\}$, this directly shows that S' is a solution vertex cover in the input instance. □

2.1.2 Caterpillar Trees with Constant Maximum Degree

After having established that Network Untangling is NP-hard on trees, we wondered whether that hardness is only due to the unrestricted maximum degree of the constructed underlying graph in the previous reduction. However, further study revealed that the problems remain NP-hard even on trees with a constant maximum degree, yielding the theorem below.

Theorem 2.4: NP-Hardness on Caterpillar Trees

MinTimeline$_\infty$ and MinTimeline$_+$ with $\ell = 0$ are NP-hard even if the underlying graph is a caterpillar tree with a maximum degree of 3.

Note that since $\ell = 0$, we again refrain from treating MinTimeline$_+$ separately and just speak of MinTimeline$_\infty$ in the following.

We show Theorem 2.4 by another reduction from Vertex Cover, which is a modification of the previous Reduction 2.3: Instead of one single vertex in the center, we use a long path to "attach" the other vertices, thereby forming a caterpillar tree with constant maximum degree. Apart from that, both reductions share similar ideas in terms of correctness. For technical reasons, we assume to be given an instance $(G = (V, E), k)$ in which the vertices are named v_1 to v_n and the edges are named e_1 to e_m.

Reduction 2.5: Let $E^* = \{e_j^* \mid j \in [m+1]\}$ and let $E^{**} = \{e_j^{**} \mid j \in [m]\}$. Let $V' = E \cup E^* \cup E^{**}$. We construct an instance $(\mathcal{G} = (V', E_1, \ldots, E_n), k', 0)$ with vertex set V' in the following way:

Set $k'_{e^*} = n - k$ for each $e \in E^*$ and set $k'_{e^{**}} = k$ for each $e \in E^{**}$. For each $e \in E$, set $k'_e = 1$.

Let $E_i = \{\{e, e^{**}\} \mid v_i \in e\} \cup \bigcup_{j \in [m]} \left\{\{e_j^*, e_j^{**}\}, \{e_j^{**}, e_{j+1}^*\}\right\}$ for each $v_i \in V$.

This means that the vertices in E^* and in E^{**} together form a path in every layer and additionally there is an edge between e_j and e_j^{**} in each layer $\mathcal{G}_i$ for which v_i is an endpoint of e_j in the input instance. Figure 2.1 shows an example of this reduction.

♦

Before proving Theorem 2.4, we introduce a lemma which highlights an important aspect of Reduction 2.5.

Depiction of Reduction 2.5

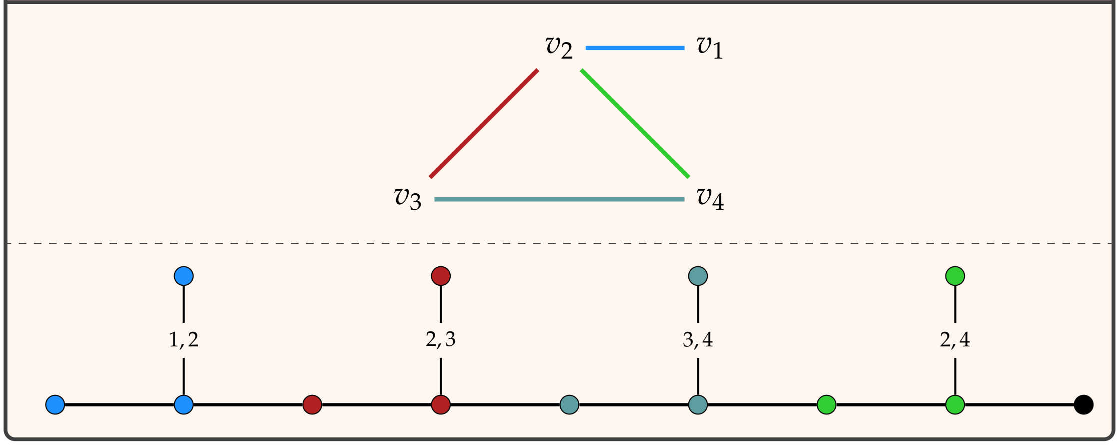

Figure 2.1: The first graph shows an input instance to Reduction 2.5 and the second graph shows the underlying graph of the corresponding output instance. For the latter, labeled edges are present in the respective layers and unlabeled edges are present in every layer. The printed colors match input instance edges to parts of the output instance created for them. For instance, the three red vertices and their specific edge-label $(2, 3)$ are constructed for the red input edge $\{v_2, v_3\}$.

Lemma 2.6: In every solution to an instance of MINTIMELINE$_\infty$ constructed by Reduction 2.5, in each layer, either all vertices in E^* or all vertices in E^{**} are active. There are $n - k$ layers where all vertices of E^* are active and in the remaining k layers all vertices of E^{**} are active.

Proof. The vertices $E^* \cup E^{**}$ form a path in the underlying graph, such that any edge $\{a, b\}$ on this path has one endpoint $a \in E^*$ and one endpoint $b \in E^{**}$. As $\{a, b\}$ appears in all n layers and its two incident vertices a and b can only be active in $k_a + k_b = (n - k) + k = n$ layers in total, each corresponding temporal edge $(\{a, b\}, t)$ has to be covered by exactly one of a or b in its respective layer $\mathcal{G}_t$. As this holds for all path edges, it proves the lemma. □

Using Reduction 2.5 and Lemma 2.6, we can now prove the theorem.

Proof of Theorem 2.4. It is easy to see that a temporal graph $\mathcal{G}$ constructed by Reduction 2.5 has $\Delta(\mathcal{G}_\downarrow) \leq 3$. We next show that the reduction's output instance has a solution for MINTIMELINE$_\infty$ if and only if the input instance has a solution for VERTEX COVER.

($\Rightarrow$) Without loss of generality, we assume the input instance has a vertex cover S of size exactly k. Then for each $v_i \in S$, we choose all vertices in E^{**} to be activated in layer $\mathcal{G}_i$ of the output instance. Note that this is possible, because $k'_{e^{**}} = k$ for each $e^{**} \in E^{**}$ and $|S| \leq k$.

With this, the only remaining uncovered temporal edges in $\mathcal{G}$ are the edges of the layers $\{\mathcal{G}_i | v_i \notin S\}$. It is easy to see that the path edges among them, i.e. edges with one endpoint in E^{**} and the other endpoint in E^*, can be covered by activating all vertices in E^* in those layers. This is again possible, because $|V \setminus S| = n - k = k'_{e^*}$ for each $e^* \in E^*$.

Each vertex $e \in E$ has at most one incident temporal edge within layers $\{\mathcal{G}_i | v_i \notin S\}$, since S is a vertex cover and by construction e has only two temporal edges, one in each layer corresponding to an endpoint of e in the input instance. Thus, all yet-uncovered temporal edges incident to any $e \in E$ can be covered by these vertices.

($\Leftarrow$) By Lemma 2.6 we know that any solution to an output instance of Reduction 2.5 has exactly k layers where all the vertices in E^{**} are active, and they are active in no other layer. Let $S \subseteq \{\mathcal{G}_1, \dots, \mathcal{G}_\tau\}$ be the set of those layers with $|S| = k$.

Any vertex $e \in E$ in the output instance has exactly two incident temporal edges, both to the same corresponding vertex $e^{**} \in E^{**}$. Let $\mathcal{G}_a$ and $\mathcal{G}_b$ be the two layers including these temporal edges. They were constructed to represent two input instance vertices v_a and v_b, such that $e = \{v_a, v_b\}$. Since both temporal edges need to be covered and $k'_e = 1$, e^{**} has to be active in at least one of $\mathcal{G}_a$ or $\mathcal{G}_b$, directly meaning that $\mathcal{G}_a \in S$ or $\mathcal{G}_b \in S$.

As $e \in E$ was chosen arbitrarily and each edge in the input instance is represented by one constructed vertex of E, it follows that S implies a vertex cover of size k in the input instance. $\square$

2.2 NP-Hardness on Bipartite Graphs with Constant Number of Layers and Maximum Degree

In the previous section, we showed that MinTimeline$_\infty$ and MinTimeline$_+$ with $\ell = 0$ are NP-hard on temporal trees. However, for both reductions we presented so far the number of layers τ of the constructed temporal graph was unbounded. Later, in Section 3.1, we will exactly find out why this unbounded τ is indeed necessary for the NP-hardness on trees. For now, we want to use this insight as motivation to explore additional graph classes on which our problems stay NP-hard—but this time while also assuming that τ is small.

For this restriction, there is already an interesting hardness result by Froese, Kunz, and Zschoche (2022), which effectively states that Network Untangling with $\ell = 0$ and $\tau = 3$ is NP-hard on all underlying graphs on which

Vertex Control Subgraph

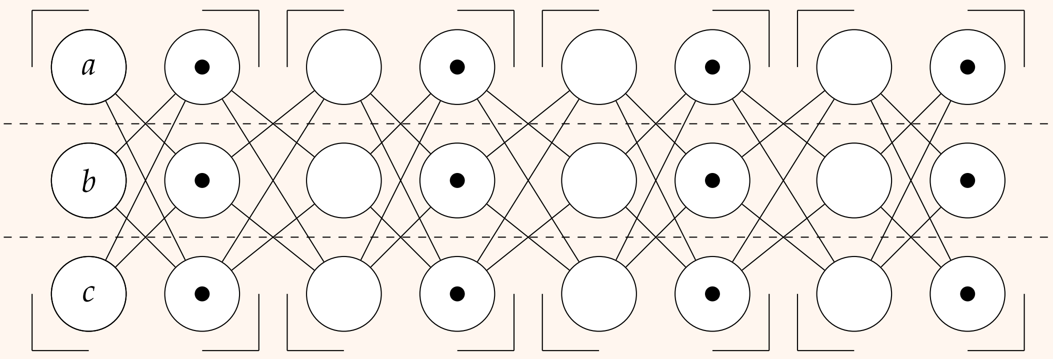

Figure 2.2: Our vertex control subgraph (used in the proof of Theorem 2.10), which can have an arbitrarily large number of 6-vertex *chunks* (shown above with four chunks for illustration purposes). As vertices a, b and c are precolored, it is easy to see that in any valid 3-coloring all vertices depicted on the same line must have the same color. Dot-marked vertices belong to the right-hand side of the bipartite graph. Using such a vertex control subgraph, every other vertex in the graph can have its color fixed or restricted by linking it to one unique chunk in an appropriate way.

the classic 3-Coloring is NP-hard as well. This for instance includes planar graphs with a constant maximum degree (Garey, Johnson, and Stockmeyer 1976). However, to our knowledge, no similar result for Network Untangling on bipartite underlying graphs with a constant number of layers was known thus far. Consequently, we researched if the restriction to bipartite temporal graphs with few layers still retains the NP-hardness. We arrived at the following theorem.

Theorem 2.7: NP-Hardness on Bipartite Graphs with $\tau = 3$

MinTimeline$_\infty$ and MinTimeline$_+$ are NP-hard even if the underlying graph of the input is bipartite, $\ell = 0$, $\tau = 3$ and $\Delta = 12$.

In order to prove the theorem, we use a reduction closely resembling the one presented by Froese, Kunz, and Zschoche (2022). In particular, they reduced from 3-Coloring to Uniform MinTimeline$_\infty$ with $\ell = 0$ by essentially copying the input graph into 3 identical layers and setting k to 2. As 3-Coloring is not NP-hard on bipartite graphs, we provide a reduction from a slightly different coloring problem instead. That problem was introduced by Biró, Hujter, and Tuza (1992) and later shown to be NP-hard on bipartite graphs (Bodlaender, Jansen, and Woeginger 1994):

Problem Definition 2.8: 1-PRECOLORING EXTENSION WITH 3 COLORS

Input: A graph $G = (V, E)$ and three special vertices $a, b, c \in V$.
Question: Is there a proper 3-coloring of G such that a, b and c receive colors 1, 2 and 3 respectively?

The requirement of these special vertices receiving particular colors effectively from the beginning is called *precoloring* of those vertices. If a vertex v is adjacent to a vertex w and w has an always fixed (e.g. precolored) color, then we call v *color-restricted* for that color, since it can not be assigned that same color.

Bodlaender, Jansen, and Woeginger (1994) did not explicitly show that 1-PRECOLORING EXTENSION is also NP-hard on bipartite graphs with a constant maximum degree, however their reduction can be slightly modified to reflect this. In order to do so, we utilize a lemma they already proved:

Lemma 2.9 (Bodlaender, Jansen, and Woeginger 1994, restated):
A graph which includes two distinct vertices x and y can be extended by introducing three disjoint paths between x and y containing three additional color-restricted vertices each, such that any proper 3-coloring of the resulting graph assigns different colors to x and y.

With this, we show the following.

Theorem 2.10

1-PRECOLORING EXTENSION WITH THREE COLORS is NP-hard on bipartite graphs even if $\Delta = 12$ and the special vertices a, b and c are on the same graph side.

Proof. We reduce from the NP-hard 3-COLORING problem on graphs with a maximum degree of four (Garey, Johnson, and Stockmeyer 1976). We assume our input graph is named $G = (V, E)$ and the output graph is named $G' = (L \cup R, E')$, where G' is bipartite with respective sides L and R.

Like Bodlaender, Jansen, and Woeginger (1994), we create a so-called vertex control subgraph in G', which includes the precolored vertices a, b and c. But unlike theirs, our vertex control subgraph contains $6 \cdot 9 \cdot |E|$ vertices, organized into groups of 6 vertices each, which we call *chunks*. The chunks are connected to one another as portayed in Figure 2.2. This way, each chunk contains exactly two known vertices of each color, one of them in L and the other in R.

We introduce a vertex v to L for each $v \in V$. For each $\{x, y\} \in E$, we introduce three paths from x to y using three vertices each as Lemma 2.9 states. We can color-restrict these path vertices as necessary by linking each of them

Outline of Reduction 2.11

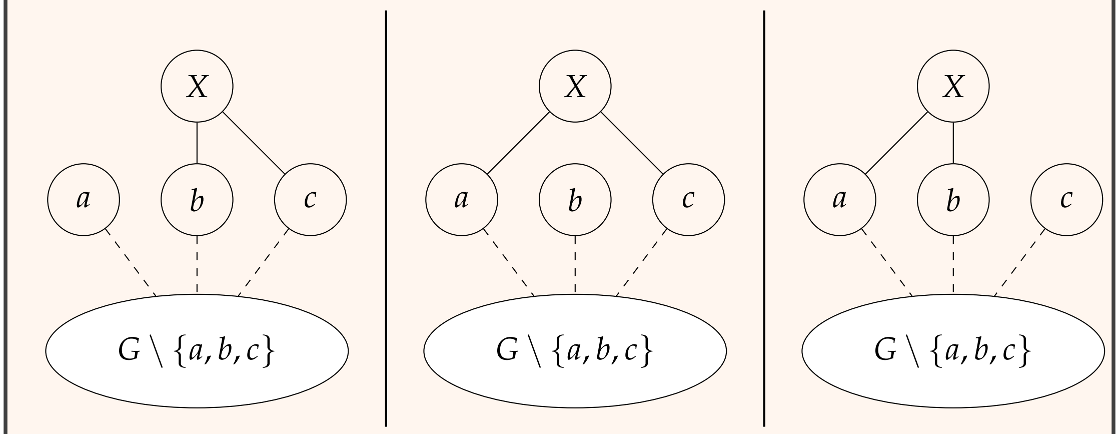

Figure 2.3: Depiction of the three layers of Reduction 2.11. All edges of the input graph are copied to every layer. All newly constructed edges are incident to X and are depicted as continuous lines.

to the respective vertex in a unique chunk of the vertex-control subgraph. Note that x and y are indeed both allowed to be in L, as those connecting paths all contain an odd number of vertices. By Lemma 2.9, we now know that x and y are assigned different colors for each $\{x, y\} \in E$, directly showing the correctness of the reduction.

We also know that a, b and c are all on the left graph side and that $\Delta(G') \leq 12$, since each edge of G was replaced by 3 paths and $\Delta(G) = 4$. The vertices in the vertex control subgraph and in the added paths clearly have degrees of less than 12. □

Using Theorem 2.10, we can next prove Theorem 2.7 by providing another Karp reduction. This time, we reduce from 1-PRECOLORING EXTENSION WITH THREE COLORS to MINTIMELINE$_\infty$. Of course, we assume the input graph G is bipartite, has $\Delta(G) = 12$, and the precolored vertices are on the same side, as Theorem 2.10 states.

Reduction 2.11: Given an input graph $G = (V, E)$ with three special vertices $a, b, c \in V$, we construct a temporal graph $\mathcal{G} = (V', E_1, E_2, E_3)$ in the following way:

- Let $V' = V \cup \{X\}$, where X is a new vertex.
- Let $E_1 = E \cup \{\{X, b\}, \{X, c\}\}$.
- Let $E_2 = E \cup \{\{X, a\}, \{X, c\}\}$.

- Let $E_3 = E \cup \{\{X, a\}, \{X, b\}\}$.

The three layers essentially just copy the input graph, except for some edges of the introduced vertex X (delineated in Figure 2.3). Together with $k_v = 2$ for each $v \in V$, $k_X = 0$ and $\ell = 0$, the temporal graph $\mathcal{G}$ then forms the output instance. ◆

Proof of Theorem 2.7. It is easy to see that the output temporal graph of Reduction 2.11 is bipartite and has a maximum degree of twelve, as long as the input is an instance satisfying the requirements of Theorem 2.10. In the following, we prove that there is a solution to the input instance if and only if there is a solution to the output instance.

Since $k_X = 0$, b and c have to be in any output solution's vertex cover of layer $\mathcal{G}_1$—which also holds for a and c in $\mathcal{G}_2$, as well as for a and b in $\mathcal{G}_3$, respectively. As $k_v = 2$ and $\tau = 3$, in any maximal solution each $v \in V$ is in the vertex cover of exactly one layer. Thus, there are three disjoint independent sets of vertices in $G' \setminus X = G$ (i.e. three disjoint sets of vertices such that $G' \setminus X$ contains no edge between two separate vertices of the same set). Using the layer numbers of the independent sets as color numbers, this directly translates (and retranslates) to G' having a proper 3-coloring with a, b, and c being assigned colors 1, 2, and 3, thereby concluding the proof. □

3 Searching for Efficiently Computable Cases Using Parameterized Methods

In this chapter, we analyze NETWORK UNTANGLING from the perspective of parameterized complexity theory. In doing so, we identify two different parameter combinations for which MINTIMELINE$_\infty$ and MINTIMELINE$_+$ are both fixed-parameter tractable and present corresponding algorithms that solve these problems (Sections 3.1 and 3.3). We also include a parameterized hardness result in Section 3.2. Finally, we derive another FPT result for NETWORK UNTANGLING by proving a relationship between two graph parameters in Section 3.4, which was previously unknown to us. For readability purposes, we point out that all algorithms in this chapter span across multiple pages and are therefore divided into several smaller parts.

3.1 FPT when Parameterized by Treewidth and Number of Layers

Previously, we established that NETWORK UNTANGLING is NP-hard on underlying trees, but that hardness result only holds for large numbers of τ, i.e. for temporal graphs with many time steps. Encouraged by this discovery, we tried to find algorithms for efficiently solving both problem variants on temporal trees if τ is small—and this time, we were successful. Moreover, we could generalize this to all temporal graphs with bounded treewidth. Therefore, in this section we present two algorithms which reveal that MINTIMELINE$_\infty$ and MINTIMELINE$_+$ are both in FPT w.r.t. $\mathbf{tw} + \tau$.

3.1.1 MinTimeline$_\infty$ Algorithm

Starting with the MINTIMELINE$_\infty$ variant, we are going to prove the following:

Theorem 3.1: MinTimeline$_\infty$ in FPT w.r.t. tw $+ \tau$

MINTIMELINE$_\infty$ is decidable in $|V| \cdot |\mathcal{G}| \cdot \left(\tau^{k_\infty} \cdot k_\infty\right)^{O(\mathbf{tw})} + 2^{O(\mathbf{tw}^3)}$ time.

Note that this implies the problem is in XP w.r.t. k_∞ and FPT w.r.t. τ on temporal graphs with bounded treewidth, the latter because $k_\infty < \tau$ for any instance of MINTIMELINE$_\infty$ which cannot be trivially reduced. In particular, if $k_v \geq \tau$ for some $v \in V$, then v is assumed to be always active and can thus be removed along with its incident temporal edges.

Algorithm 3.1: FPT w.r.t. **tw** $+ \tau$ for MINTIMELINE$_\infty$ (Part 1)

```
Data: A temporal graph $\mathcal{G} = (V, E_1, \ldots, E_\tau)$, a number $\ell \in \mathbb{N}$, an integer
      vector $k \in \mathbb{N}^{|V|}$ and a tree decomposition $D$ of $\mathcal{G}_\downarrow$ with set of bags $B$.
      We assume $D$ to be rooted in a bag $r \in B$, meaning that each bag
      other than $r$ has a unique parent bag.
/* The high-level goal of the algorithm is to fill its array $T$
   in the form of a dynamic program over the given tree
   decomposition $D$.  In order to do so, child bags have to be
   processed before their parents (i.e. using bottom-up
   traversal).  Finally, the instance is YES if the root bag
   has a non-empty array entry.                              */
1 procedure main ()
2     Initialize globally-accessible array $T$ with $T[b] = \emptyset$ for each $b \in B$
3     foreach bag $b \in B$ in bottom-up-order do process_bag($b$)
4     if $T[r]$ is $\emptyset$ then return NO else return YES
5 end procedure
```

The corresponding Algorithm 3.1 solves MINTIMELINE$_\infty$ with a rather typical dynamic programming approach on a precomputed tree decomposition of the underlying graph of an input instance. Traversing the tree from the leafs upwards, at each bag, each possible way of activating the bag's vertices across the τ layers is tested out, specifically by checking if it forms a local solution. This is mainly what results in the high dependency of the running time on **tw** and τ. Successful test results are written in a table and later revisited at the respective parent bags in order to verify that these local solutions can be properly extended.

Algorithm 3.1: FPT w.r.t. **tw** $+ \tau$ for MinTimeline$_\infty$ (Part 2)

```
/* process_bag finds all possible vertex occurrence
   combinations for a given bag with regards to its descendants
   (see Lemma 3.2 for details). They are then stored in the
   array T.                                                    */
6  procedure process_bag ($b \in B$)
7      foreach function $f : b \to \mathcal{P}([\tau])$ with $|f(v)| \le k_v$ for each $v \in b$ do
8          if check_covering($b, f, b, f$) is false then continue
9          store ← true
10         foreach child bag $c$ of $b$ in $D$ do
11             if there is no function $g$ in $T[c]$ such that
                check_covering($b, f, c, g$) is true then store ← false
12         end foreach
13         if store is true then $T[b] \leftarrow T[b] \cup \{f\}$
14     end foreach
15 end procedure
/* Given two vertex sets S and Q and respective vertex
   occurrences for those sets, check_covering checks if all
   temporal edges with endpoints both in S and in Q are
   correctly covered, assuming all their vertices are activated
   at the corresponding given time steps.                      */
16 procedure check_covering ($S \subseteq V, f : S \to \mathcal{P}([\tau]), Q \subseteq V,$
    $g : Q \to \mathcal{P}([\tau])$)
17     foreach $v \in S \cap Q$ do
18         if $f(v) \neq g(v)$ then return false
19     end foreach
20     foreach temporal edge $e = (\{v, w\}, t)$ in $\mathcal{G}$ with $v \in S$ and $w \in Q$ do
21         if $f(v) \cup g(w)$ contains no $t'$ such that $t' \le t \le t' + \ell$ then
22             return false
23         end if
24     end foreach
25     return true
26 end procedure
```

In order to show Theorem 3.1 using Algorithm 3.1, we initially state two lemmas about this algorithm.

Lemma 3.2: Let $\mathcal{G}_b$ be the subgraph of $\mathcal{G}$ induced by the vertices in bag b and its descendant bags in D. Let k' be the corresponding subvector of k, i.e. the unique vector with only $k'_v = k_v$ for each $v \in \mathcal{G}_b$. A function f is saved in $T[b]$ by the procedure `process_bag(b)` of Algorithm 3.1 if and only if there is a solution to the instance $(\mathcal{G}_b, k', \ell)$ in which each $v \in b$ is activated at time steps $f(v)$.

Proof. In order for a function f to be saved in $T[b]$, the function has to pass several calls to `check_covering`. The first call in Line 8 ensures that each temporal edge between vertices in b is covered by activating each $v \in b$ at the time steps in $f(v)$. The other calls (Line 11) ensure that for each child c of b in D, there is a saved function g in $T[c]$ such that all temporal edges between vertices in b and c can be covered by activating each $v \in b$ at times $f(v)$ and each $w \in c$ at times $g(w)$.

Note that each vertex v is activated for at most k_v times by each function (see Line 7) and two functions f and g can only pass `check_covering` if they map same vertices to same time step sets. By induction over the descendants of b in D (which were processed before b), this means that all remaining temporal edges of $\mathcal{G}_b$ can be covered after activating each $v \in b$ within layers $f(v)$, while still activating each $u \in \mathcal{G}_b$ only for at most k_u times. $\square$

Lemma 3.3: Let $k' = \min(k_\infty, \frac{\tau}{2})$. Algorithm 3.1 runs in time $O\left(\left(\binom{\tau}{k'} \cdot k_\infty + 1\right)^{2\mathbf{tw}+2} \cdot |V| \cdot |\mathcal{G}|\right)$, when given a tree decomposition D with minimum treewidth.

Proof. For every bag b, the algorithm tries out at most $\left(\binom{\tau}{k'} \cdot k_\infty + 1\right)^{\mathbf{tw}+1}$ functions (Line 7) and compares each of them against at most the same number of functions of every child bag of b. This number results from having at most $1 + \sum\limits_{i \in [k_v]} \binom{\tau}{i} \leq 1 + k_\infty \binom{\tau}{k'}$ distinct ways to map each $v \in b$ individually and having at most $\mathbf{tw} + 1$ vertices in b.

The `check_covering` procedure called for each such combination takes time $O(|\mathcal{G}|)$. Every bag is only processed once and its functions are only compared against functions of at most one parent bag. We assume that D contains fewer than $|V|$ bags, because otherwise it can be transformed in $O(|V| \cdot |\mathcal{G}|)$ time into such a tree decomposition without changing the treewidth (Bodlaender 1996).

Thus, the total running time is in $O\left(\left(\binom{\tau}{k'}\cdot k_\infty+1\right)^{2\mathbf{tw}+2}\cdot|V|\cdot|\mathcal{G}|\right)$. □

With this knowledge, we can now prove Theorem 3.1.

Proof of Theorem 3.1. By choosing b to be the root bag r in Lemma 3.2, we directly infer that Algorithm 3.1 saves at least one function in $T[r]$ if the input is a YES-instance and does not save a function there otherwise. The algorithm then returns YES or NO accordingly (see Line 4). Lemma 3.3 proves the algorithm's running time, if we previously use the algorithm of Bodlaender (1996) to compute a minimum tree decomposition D of $\mathcal{G}_\downarrow$. □

3.1.2 MinTimeline$_+$ Algorithm

Having addressed MinTimeline$_\infty$ in the previous section, we now want to provide a similar theorem for MinTimeline$_+$:

Theorem 3.4: MinTimeline$_+$ in FPT w.r.t. $\mathbf{tw}+\tau$

MinTimeline$_+$ can be solved in time $|\mathcal{G}|\cdot|V|\cdot\min\left(2^{O(\mathbf{tw}\cdot\tau)},(\mathbf{tw}\cdot\tau)^{O(\mathbf{tw}\cdot k_\infty+\ell)}\right)+2^{O(\mathbf{tw}^3)}\cdot|V|$.

This implies that MinTimeline$_+$ is also in XP w.r.t. $k_\infty+\ell$ and in FPT w.r.t. τ on temporal graphs with constant treewidth.

Algorithm 3.2 uses a method similar to the previous Algorithm 3.1 for solving MinTimeline$_+$, i.e. it is based on a dynamic program on a given tree decomposition. We again try out all possible ways to activate vertices of each individual bag and construct a global solution step by step from these local ones. Since this problem variant additionally imposes a limit on the total sum of interval lengths of any global solution, we additionally record the sum of interval lengths if each local solution in our dynamic programming table.

Algorithm 3.2: FPT w.r.t. **tw** $+ \tau$ for MINTIMELINE$_+$ (Part 1)

```
Data: A temporal graph $\mathcal{G} = (V, E_1, \dots, E_\tau)$, a number $\ell \in \mathbb{N}$, an integer
      vector $k \in \mathbb{N}^{|V|}$ and a tree decomposition $D$ of $\mathcal{G}_\downarrow$ with set of bags $B$.
      We assume $D$ to be rooted in a bag $r \in B$, meaning that each bag
      other than $r$ has a unique parent bag.
/* Similar to the corresponding routine of Algorithm 3.1.                    */
procedure main ()
    Initialize globally-accessible array $T$ with $T[b] = \emptyset$ for each $b \in B$
    foreach bag $b \in B$ in bottom-up-order do process_bag($b$)
    if $T[r]$ is $\emptyset$ then return NO else return YES
end procedure
/* Modified for this problem variant from Algorithm 3.1.
   Additionally computes and stores an interval price for each
   saved function which is not allowed to exceed $\ell$.                     */
procedure process_bag ($b \in B$)
    foreach function $f : b \to \mathcal{P}([\tau])$ with $\sum_{v \in b} |f(v)| \leq \sum_{v \in b} k_v + \ell$ do
        if check_covering($b, f, b, f$) is false then continue
        $y \leftarrow$ compute_sum($b, f$)
        foreach child bag $c$ of $b$ in $D$ do
            let $(g, x) \in T[c]$ such that check_covering($b, f, c, g$) is true and $x$
             is minimal
            if $g$ is undefined then $y \leftarrow \ell + 1$
            else $y \leftarrow y + x -$ compute_sum($c \cap b, f$)
        end foreach
        if $y \leq \ell$ then $T[b] \leftarrow T[b] \cup \{(f, y)\}$
    end foreach
end procedure
```

Algorithm 3.2: FPT w.r.t. **tw** $+ \tau$ for MinTimeline$_+$ (Part 2)

```
/* A greedy sub-algorithm to calculate the necessary interval
   price for realizing the given vertex occurrences of a set.
   */
procedure compute_sum ($S \subseteq V, f : S \to \mathcal{P}([\tau])$)
    result $\leftarrow 0$
    foreach $v \in S$ do
        if $k_v$ is zero and $f(v)$ is not $\emptyset$ then return $\ell + 1$
        $\mathcal{T} \leftarrow f(v)$
        while $|\mathcal{T}| > k_v$ do
            let $t' \in f(v)$ and $t \in \mathcal{T}$ such that $t' < t$ and $t - t'$ is minimal
            result $\leftarrow$ result $+ (t - t')$
            $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$
        end while
    end foreach
    return result
end procedure
/* Similar to the respective procedure in Algorithm 3.1, except
   it does not need to account for vertex intervals.          */
procedure check_covering ($S \subseteq V, f : S \to \mathcal{P}([\tau]), Q \subseteq V,$
  $g : Q \to \mathcal{P}([\tau])$)
    foreach $v \in S \cap Q$ do
        if $f(v) \neq g(v)$ then return false
    end foreach
    foreach temporal edge $e = (\{v, w\}, t)$ in $\mathcal{G}$ with $v \in S$ and $w \in Q$ do
        if $t \notin f(v) \cup g(w)$ then return false
    end foreach
    return true
end procedure
```

For the purpose of showing Theorem 3.4 with Algorithm 3.2 we again make use of various lemmas.

Lemma 3.5: Assume we have an instance of MINTIMELINE$_+$ as described by Algorithm 3.2. Let $S \subseteq V$ such that $k_v > 0$ for each $v \in S$ and let $f : S \to \mathcal{P}([\tau])$. For each $v \in S$, let L_v be the minimum number $x \in \mathbb{N}$ of time steps where v has to be active but not activated in order to be active at the time steps $f(v)$ while using only k_v activations. The procedure `compute_sum`(S, f) of Algorithm 3.2 outputs $\sum\limits_{v \in S} L_v$.

Proof. The procedure handles each $v \in S$ separately and sums up the individual L_v. We assume that $|f(v)| > k_v$, as otherwise L_v is trivially zero. Each time step $b \in f(v)$ at which v is active but not activated increases L_v by the length of the shortest interval from any other $a \in f(v)$ to b.

In this sense, the time steps in $f(v)$ effectively only differ by their interval lengths and are otherwise exchangeable. In order to minimize L_v, `compute_sum` can thus select the time steps at which v should not be activated using a greedy strategy, i.e. by repeatedly taking the shortest still available interval between any distinct $a, b \in f(v)$ (marking b as time step where v is active but not activated), until only k_v activations of v remain. □

Lemma 3.6: Let $\mathcal{G}_b$ be the subgraph of $\mathcal{G}$ induced by the vertices in bag b and its descendant bags in D. Let k' be the corresponding subvector of k, i.e. the unique vector with only $k'_v = k_v$ for each $v \in \mathcal{G}_b$. A function f is saved in $T[b]$ by the procedure `process_bag(b)` of Algorithm 3.2 if and only if there is a solution to the instance $(\mathcal{G}_b, k', \ell)$ in which each $v \in b$ is activated at time steps $f(v)$.

Proof. The proof is nearly identical to the proof of Lemma 3.2 —except this time we additionally need to keep track of the minimum number $x \leq \ell$ for each saved function f, such that the instance $(\mathcal{G}_b, k', x)$ has a corresponding solution using f on the vertices of b. This way, we can ensure that x never exceeds ℓ for a saved function. We call x the function's *interval price*.

In order to compute that price for each function f, we use the procedure `compute_sum` on bag b with f as described by Lemma 3.5 (if $k_v = 0$ and $f(v) \neq \emptyset$ for some $v \in b$, then `compute_sum` signifies that there is no respective solution by returning $\ell + 1$). Additionally, we add the interval prices of the used functions of each child c of b to the interval price of f (counting vertices in $c \cap b$ only once), so that the saved interval price of f on bag b reflects the lowest possible total price of the sub-instance that the lemma refers to. □

Lemma 3.7: Let $p = (\mathbf{tw}+1)\cdot k_\infty + \ell$. Algorithm 3.2 runs in time $O\left(\tau \cdot |\mathcal{G}| \cdot |V| \cdot \min\left(2^{(\mathbf{tw}+1)\cdot\tau}, p\cdot((\mathbf{tw}+1)\tau)^p\right)^2\right)$ when given a tree decomposition D with minimum treewidth.

Proof. The set of all vertex occurences including vertices of a bag b has size at most $(\mathbf{tw}+1)\cdot\tau$. At Line 7, the algorithm iterates over all its subsets with size at most p. This yields at most $\min(2^{(\mathbf{tw}+1)\cdot\tau}, p\cdot((\mathbf{tw}+1)\tau)^p)$ such subsets, limiting the number of functions which are checked per bag.

Each such function of a bag b is compared against at most the same number of saved functions of each child bag of b and these comparisons (i.e. calling the `check_covering` and `compute_sum` procedure for each function pair) take at most $O(|\mathcal{G}|\cdot\tau)$ time each. This results in the above running time, since we assume that D has at most $|V|$ bags (otherwise D can be accordingly modified beforehand using a procedure described by Bodlaender (1996)). □

Using these lemmas, we can prove Theorem 3.4.

Proof of Theorem 3.4. By choosing b to be the root bag r in Lemma 3.6, we directly infer that Algorithm 3.2 saves at least one function in $T[r]$ if the input is a YES-instance and does not save a function there otherwise. The algorithm then returns YES or NO accordingly (see Line 4). Adapting Lemma 3.7 shows the stated running time, assuming we previously use the method described by Bodlaender (1996) to compute a tree decomposition D of minimum treewidth. □

3.2 W[2]-Hardness w.r.t. $k_\infty + \ell$ on Temporal Graphs with dlf $= 1$

In the previous section, we learned that both introduced variants of NETWORK UNTANGLING are fixed-parameter tractable for the combined parameter $\mathbf{tw}+\tau$. A natural question to ask would be if these results can be improved by replacing the bound on the number of layers τ with a bound on $k_\infty+\ell$, because at least regarding the MINTIMELINE$_\infty$ problem we know that $k_\infty+\ell<\tau$ for any non-reducible instance. However, we have to answer this question in the negative, as the following theorem indicates.

Theorem 3.8: W[2]-Hardness w.r.t. dlf $+ k_\infty + \ell$

MINTIMELINE$_+$ and MINTIMELINE$_\infty$ are both W[2]-hard w.r.t. k_∞, even if the underlying graph's distance to a linear forest is one and $\ell = 0$.

Illustration of Reduction 3.10

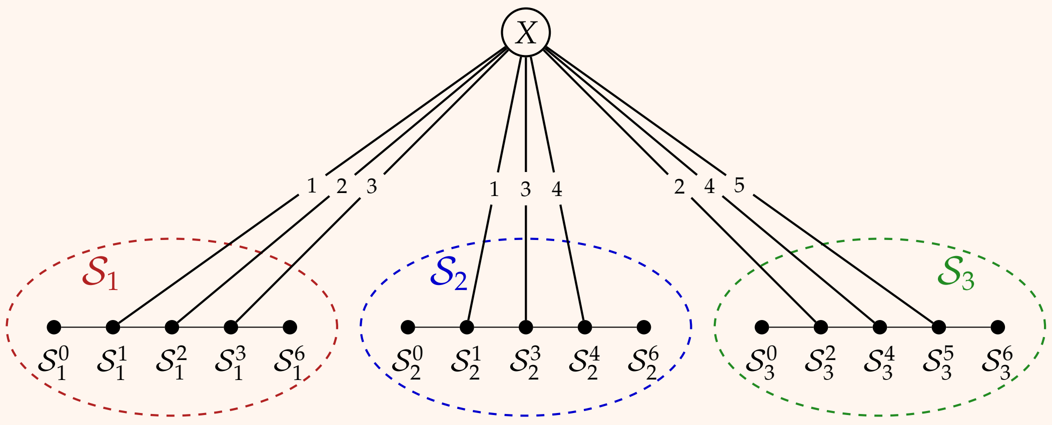
X
1 2 3
1 3 4
2 4 5
$\mathcal{S}_1$
$\mathcal{S}_2$
$\mathcal{S}_3$
$\mathcal{S}_1^0$ $\mathcal{S}_1^1$ $\mathcal{S}_1^2$ $\mathcal{S}_1^3$ $\mathcal{S}_1^6$
$\mathcal{S}_2^0$ $\mathcal{S}_2^1$ $\mathcal{S}_2^3$ $\mathcal{S}_2^4$ $\mathcal{S}_2^6$
$\mathcal{S}_3^0$ $\mathcal{S}_3^2$ $\mathcal{S}_3^4$ $\mathcal{S}_3^5$ $\mathcal{S}_3^6$

Figure 3.1: Output temporal graph of Reduction 3.10 for the HITTING SET instance $(U, (S_1, S_2, S_3), k) = ([5], (\{1, 2, 3\}, \{1, 3, 4\}, \{2, 4, 5\}), k)$. The edges annotated with numbers are present in that numbered layer. All non-annotated edges are present within a unique layer only containing that single temporal edge. The number k in the input instance is only relevant for setting the vector entry k'_X in the constructed instance. For each other constructed vertex v with underlying degree 3, we set $k'_v = 2$. If the underlying degree of v is 1 instead, then we set $k'_v = 0$.

Note that W[2]-hardness w.r.t. $\mathbf{tw} + k_\infty + \ell$ follows from the theorem, as $\mathbf{tw}(G) \leq \mathbf{dlf}(G) + 1$ for each simple graph G. We next show Theorem 3.8 by providing an appropriate FPT-reduction from HITTING SET, which is W[2]-hard w.r.t. its parameter k (Downey and Fellows 1999).

Problem Definition 3.9: HITTING SET

Input: A finite set $U \subseteq \mathbb{N}$, a collection of subsets $S_1, \ldots, S_m \subseteq U$ and a number $k \in \mathbb{N}$.
Question: Is there an $H \subseteq U$ with $|H| \leq k$ and $H \cap S_j \neq \emptyset$ for all $j \in [m]$?

Reduction 3.10: Given an instance $(U, (S_1, \ldots, S_m), k)$ of HITTING SET we construct a temporal graph $\mathcal{G}$ in the following way.

For each given set S_j, create a set $\mathcal{S}_j$ containing $|S_j| + 2$ new vertices. Formally, let $\mathcal{S}_j = \bigcup_{u \in S_j} \left\{\mathcal{S}_j^u\right\} \cup \left\{\mathcal{S}_j^0, \mathcal{S}_j^{m+1}\right\}$ for each $j \in [m]$. Let X be a new vertex and let $V = \bigcup_{j \in [m]} \mathcal{S}_j \cup \{X\}$. We use V as the vertex set of $\mathcal{G}$. For each $u \in U$, let the layer $\mathcal{G}_u$ have the edge set $E_u = \Big\{\{X, \mathcal{S}_j^u\} \mid \mathcal{S}_j^u \in V \setminus \{X\}\Big\}$.

For each $j \in [m]$, for each $\mathcal{S}_j^u \in \mathcal{S}_j \setminus \{\mathcal{S}_j^{m+1}\}$, let $\mathcal{S}_j^v \in \mathcal{S}_j$ with $u < v$ so that there is no $\mathcal{S}_j^w \in \mathcal{S}_j$ with $u < w < v$. With each such pair of $\mathcal{S}_j^u$ and $\mathcal{S}_j^v$ create a new unique layer containing only an edge between these two vertices. This way, the constructed underlying graph contains a path subgraph from $\mathcal{S}_j^0$ to $\mathcal{S}_j^{m+1}$ for each $\mathcal{S}_j$. An example for this construction is illustrated in Figure 3.1.

To construct a vector k' for MINTIMELINE$_\infty$, set $k'_X = k$. Afterwards, set $k'_{\mathcal{S}_j^0}$ and $k'_{\mathcal{S}_j^{m+1}}$ to zero for each $j \in [m]$ and set $k'_v = 2$ for all remaining vertices $v \in V$. Together with $\ell = 0$, $\mathcal{G}$ and k' form the output instance. ⧫

Using Reduction 3.10 we can directly prove the previous theorem.

Proof of Theorem 3.8. First note that the order of created layers in Reduction 3.10 does not matter except for description purposes, because $\ell = 0$. To prove the reduction's correctness, we show that its output is a YES-instance of MINTIMELINE$_\infty$ if and only if its input was a YES-instance of HITTING SET. This is sufficient to prove the theorem, as the reduction runs in polynomial time, $k'_\infty = k$, and only the vertex X needs to be deleted for the underlying graph of any constructed instance to become a linear forest.

($\Rightarrow$) If the input instance has a solution H, then we build up a solution to the constructed instance by first using X in the vertex cover of the corresponding layers $\{\mathcal{G}_u \mid u \in H\}$. Since X is an endpoint of all edges in those layers, no other node needs to be activated there. For each $j \in [m]$, there is now at least one $v \in \mathcal{S}_j$ with an incident temporal edge covered by X, because H is a hitting set solution. As v has only three temporal edges in total (one to X and two on the path from $\mathcal{S}_j^0$ to $\mathcal{S}_j^{m+1}$) and $k'_v = 2$, v can be activated to cover its other adjacent temporal edges. It is then easily seen that we can cover so-far uncovered temporal edges using the vertices $\mathcal{S}_j \setminus \{v, \mathcal{S}_j^0, \mathcal{S}_j^{m+1}\}$. Specifically, those nodes cover either their temporal edge closer to $\mathcal{S}_j^0$ if they are on the path from v to $\mathcal{S}_j^0$ or their temporal edge closer to $\mathcal{S}_j^{m+1}$ if they are on the path from v to $\mathcal{S}_j^{m+1}$. Additionally, they all cover a temporal edge to the vertex X if otherwise uncovered. This is possible, because each vertex apart from X is activated at most two times this way.

($\Leftarrow$) If the input is a NO-instance, then each $H' \subseteq U$ with $|H'| \leq k$ has at least one set S_j where $S_j \cap H = \emptyset$. Since $k'_X = k$, this means that for every potential solution of the output instance there is at least one $\mathcal{S}_j$ where no $v \in \mathcal{S}_j$

has an incident temporal edge covered by X. The number of temporal edges incident to vertices in $\mathcal{S}_j$ is $(|S_j| + 1) + |S_j| = 2 \cdot |S_j| + 1$, since they consist of an underlying path of $|S_j| + 2$ vertices (with each edge being present in just one layer) and $|S_j|$ additional temporal edges to X. Each of those temporal edges lies in a distinct layer. But the maximum number of layers where vertices of $\mathcal{S}_j$ can be active is only $2 \cdot |S_j|$, because $k'_{\mathcal{S}_j^i} = 2$ for each $i \in S_j$, $k'_{\mathcal{S}_j^0} = k'_{\mathcal{S}_j^{m+1}} = 0$, and $\ell = 0$. Thus, at least one temporal edge cannot be covered and the output consequently is a NO-instance of MINTIMELINE$_\infty$. □

3.3 FPT when Parameterized by Tree Partition Width, k_∞ and ℓ

Knowing that an FPT-algorithm for NETWORK UNTANGLING with the combined parameter $\mathbf{tw} + k_\infty + \ell$ does not exist unless FPT $=$ W[2], we realize that we should try finding FPT-algorithms for larger parameters. Of course, we still want to keep the parameterization as small as possible in terms of parameter relationships. Additionally, we are still interested in cases where $k_\infty + \ell$ is small, since we handled the cases with bounded τ and **tw** in Section 3.1 already. Thus, we looked for a graph parameter that is hopefully only slightly larger than treewidth, but is not upper-bounded by the distance to a linear forest (because of Theorem 3.8). We arrived at the tree partition width parameter. Using **tpw**, we were successful in formulating parameterized algorithms for our problems in cases where $k_\infty + \ell$ is small. The current section is dedicated to delineating these algorithms.

3.3.1 MinTimeline$_\infty$ Algorithm

Beginning with the MINTIMELINE$_\infty$ problem, we first observe the following:

Observation 3.11: If the number of temporal edges between two distinct vertices a and b exceeds $(k_a + k_b) \cdot (\ell + 1)$ in an instance of MINTIMELINE$_\infty$, then it is a NO-instance.

Proof. With a single activation of the vertex a at most $\ell + 1$ of those temporal edges can be covered and the same holds for vertex b. Since a and b can be activated for $(k_a + k_b)$ times in total, $(k_a + k_b) \cdot (\ell + 1)$ upper-bounds the number of temporal edges with endpoints a and b that can be covered. □

From now on, because of Observation 3.11, we assume to be given instances of MINTIMELINE$_\infty$ where each edge is present in at most $2k_\infty \cdot (\ell + 1)$ layers.

With that, we are now ready to state this part's main theorem.

Theorem 3.12: MinTimeline$_\infty$ in FPT w.r.t. $\mathbf{tpw} + k_\infty + \ell$

MINTIMELINE$_\infty$ can be solved in time $(k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2)^{O(k_\infty^2 \cdot \mathbf{tpw}^2)} \cdot |\mathcal{G}| \cdot |V|$ when given a tree partition of $\mathcal{G}_\downarrow$ of width $\mathbf{tpw}$.

It follows that the problem is in FPT w.r.t. $\mathbf{tpw} + k_\infty + \ell$, since a tree partition of minimum width can be polynomially approximated in polynomial time (Bodlaender, Groenland, and Jacob 2022).

Algorithm 3.3: FPT w.r.t. $\mathbf{tpw} + k_\infty + \ell$ for MINTIMELINE$_\infty$ (Part 1)

```
Data: A temporal graph $\mathcal{G} = (V, E_1, \ldots, E_\tau)$, a number $\ell \in \mathbb{N}$, an integer
      vector $k \in \mathbb{N}^{|V|}$ and a tree partition $D$ of $\mathcal{G}_\downarrow$ with set of bags $B$. We
      assume $D$ to be rooted in a bag $r \in B$, meaning that each bag other
      than $r$ has a unique parent bag.
/* The high-level goal of the algorithm is to fill its array $T$
   in the form of a dynamic program over the given tree
   partition $D$.  In order to do so, child bags have to be
   processed before their parents (i.e. using bottom-up
   traversal).  Finally, the instance is YES if the root bag
   has a non-empty array entry.                                  */
procedure main ()
    Initialize globally-accessible array $T$ with $T[b] = \emptyset$ for each $b \in B$
    foreach bag $b \in B$ in bottom-up-order do process_bag($b$)
    if $T[r]$ is $\emptyset$ then return NO else return YES
end procedure
/* process_bag finds all possible vertex occurrences for a
   given bag with regards to its descendants (see Lemma 3.13
   for details).  They are then stored in the array $T$.          */
// parent : $B \to B \cup \emptyset$ denotes a bag's parent bag in $D$ (yields $\emptyset$
   for root bag).
// children : $B \to \mathcal{P}(B)$ denotes set of a bag's child bags in $D$.
procedure process_bag ($b \in B$)
    $\mathcal{T} \leftarrow \{t \mid (\{a,b\},t) \in \mathcal{G}, \{a,b\} \subseteq (b \cup \text{parent}(b))\}$
    foreach function $f : b \to \mathcal{P}(\mathcal{T})$ with $|f(v)| \le k_v$ for each $v \in b$ do
        if check_covering($b, f, b, f$) $\neq \emptyset$ then continue
        if distribute($b, f$, children($b$)) is false then continue
        $T[b] \leftarrow T[b] \cup \{f\}$
    end foreach
end procedure
```

Algorithm 3.3: FPT w.r.t. **tpw** $+ k_\infty + \ell$ for MINTIMELINE$_\infty$ (Part 2)

```
/* This function is the main addition when compared to the
   previous Algorithm 3.1.  It is a bounded search tree
   algorithm to check if a given bag $b$'s function $f$ can be
   extended such that all temporal edges to child bags (given
   as the set $C$) are covered as well.  This is also why the
   function recursively calls itself.                          */
procedure distribute $(b \in B, f : b \to \mathcal{P}([\tau]), C \subseteq B)$
    foreach $v \in b$ do
        if $|f(v)| > k_v$ then return false
    end foreach
    if $C = \emptyset$ then return true
    choose any $c \in C$
    foreach $g \in T[c]$ do
        if check_covering $(b, f, c, g) = \emptyset$ then
            return distribute $(b, f, C \setminus \{c\})$
        end if
    end foreach
    foreach $g \in T[c]$ do
        $f' \leftarrow$ extend $(b, f,$ check_covering $(b, f, c, g))$
        if distribute $(b, f', C \setminus \{c\})$ is true then return true
    end foreach
    return false
end procedure
```

The presented Algorithm 3.3 decides MINTIMELINE$_\infty$ in FPT-time w.r.t. **tpw** $+ k_\infty + \ell$ by interleaving the dynamic programming method already known from Algorithm 3.1 with an additional bounded search tree paradigm. The main reason for this change is that we do not parameterize by τ as we did in the setting of the previous two algorithms, and consequently cannot try out all possible ways to activate the vertices of a particular bag. Instead, we limit these checked vertex activations per bag to a number of layers depending only on **tpw** $+ k_\infty + \ell$. We specifically use the layers where we have temporal edges between nodes of the bag itself when joined with its parent bag.

For each bag, the algorithm subsequently allocates the remaining possible activations of its vertices with a bounded search tree approach, so that it can cover otherwise uncovered temporal edges to vertices in child bags. Technically, this is also why we need to use a tree partition instead of the previous tree decomposition as the foundation of this dynamic program (as Theorem 3.8 further demonstrated): A tree partition guarantees that all neighbors of each vertex are either in the same bag as the vertex itself or in an adjacent bag, which in our case ensures that we can indeed construct a global solution from the local solutions that we find by utilizing bounded search trees.

In order to prove Theorem 3.12 based on Algorithm 3.3, we first introduce some lemmas, as usual.

Lemma 3.13: Let $\mathcal{G}_b$ be the subgraph of $\mathcal{G}$ induced by vertices in bag b and its descendant bags of the tree partition D. Let k' be the corresponding subvector of k, i.e. the unique vector with only $k'_v = k_v$ for each $v \in \mathcal{G}_b$. Let p be the parent bag of b in D, or $p = \emptyset$ if b is the root. Let $\mathcal{T} \subseteq [\tau]$ be the time steps where temporal edges between vertices in $b \cup p$ are present. The procedure `process_bag(b)` of Algorithm 3.3 saves every function $f : b \to \mathcal{P}(\mathcal{T})$ in $T[b]$, for which there is a solution to the instance $(\mathcal{G}_b, k', \ell)$ where each $v \in b$ is activated at time steps $f'(v) \supseteq f(v)$.

Proof. Clearly, the algorithm examines all such functions (Line 8). For each function f which gets saved in $T[b]$, a call to `check_covering` in Line 9 ensures that each temporal edge between vertices in b is covered when each $v \in b$ is activated at the time steps in $f(v)$.

Let C be the set of child bags of b in D. As D is a tree partition, all remaining temporal edges in $\mathcal{G}_b$ that any $v \in b$ potentially has to cover are between itself and vertices in C.

For each $c \in C$, let $\mathcal{T}_c \subseteq [\tau]$ be the set of time steps where temporal edges between vertices in $b \cup c$ are present. By induction of the lemma, for each child bag $c \in C$, we previously saved every function $g : c \to \mathcal{P}(\mathcal{T}_c)$ in $T[c]$ such that a solution to the instance of the temporal graph $\mathcal{G}_c$ with a corresponding subvector of k' and ℓ can be constructed, where each $v \in c$ is activated at a set of time steps including $g(v)$. We know that all bags in $C \cup \{b\}$ are pairwise disjoint, since D is a tree partition. This especially means that if we pick one function for each one of those bags, using the bag as the function's domain, then no vertex is mapped by multiple picked functions.

Thus, it remains to check if there is a set of functions $G = \{g_c \mid c \in C\}$ where each $g_c \in G$ is saved in $T[c]$ and where we can find a function $f' : b \to \mathcal{P}(\tau)$ with $f(v) \subseteq f'(v)$ and $|f'(v)| \leq k_v$ for each $v \in b$, such that for each $c \in C$ every temporal edge between any $w \in c$ and any $v \in b$ is covered if we activate w at time steps $g_c(w)$ and v at time steps $f'(v)$. We call a function $f' : b \to \mathcal{P}(\tau)$ with $f(v) \subseteq f'(v)$ for all $v \in b$ an *extension* of the function f.

Clearly, if a solution exists and the correct set G is known, then f' can be directly constructed from f by covering otherwise uncovered temporal edges to vertices in child bags using the vertices in b. For this reason, the `distribute` routine (Line 14), when called by `process_bag`, recursively goes through all

$c \in C$ and searches for G by looking at each $g_c \in T[c]$.

If there are no uncovered temporal edges between vertices of b and the current $c \in C$ using the intermediate functions f' for b and g_c for c, then it is clearly correct to pick g_c and to avoid trying out all other saved functions for c (see Line 21). Otherwise, for each g_c saved in $T[c]$, the procedure checks if f can be extended to an intermediate f' such that activating each $w \in c$ at time steps $g_c(w)$ and activating each $v \in b$ at time steps $f'(v)$ covers all temporal edges between v and w. If yes, then the next recursive call continues this branch by looking at the next child bag in C. If not, then the current branch can not lead to a correct G and is therefore discarded. Such a stepwise extension of f to f' is possible, since each intermediate f' is given to the subsequent recursive call and the function extensions are monotone in the sense that they never make a set of time steps a particular vertex is mapped to lose elements.

Note that `distribute` ensures that each vertex v is activated for at most k_v times when applying each function, even if extended (see Line 16). By induction over the descendants of b in D, this means that all temporal edges in $\mathcal{G}_b$ can be covered after activating each $v \in b$ in layers $f'(v)$, while still activating each $u \in \mathcal{G}_b$ only for at most k_u times. □

Lemma 3.14: The procedure `distribute`(b, f, C) (see Line 14) runs in time $O(x^{k_\infty \cdot \mathbf{tpw}} \cdot (|C| + 1) \cdot |\mathcal{G}|)$, where $x = \max_{c \in C} |T[c]|$, whenever called by the `process_bag` procedure.

Proof. Consider an arbitrary call of the `distribute` procedure and let $c \in C$ be the child of b chosen in this call.

Assume that there is a function $g \in T[c]$ such that the check in Line 21 succeeds. Then, the algorithm does not branch and returns `distribute`$(b, f, C \setminus \{c\})$. The recursion depth of these `distribute` calls is at most $|C| + 1$, as with each subsequent call one element is removed from C and $C = \emptyset$ is a base case. Every iteration looks at up to x functions and calls `check_covering` for each of them, and checks the base cases, so in total all calls of this case take at most $O(x \cdot (|C| + 1) \cdot |\mathcal{G}|)$ time.

Let us now consider the other case, i.e. that there is no function $g \in T[c]$ such that `check_covering`$(b, f, c, g) = \emptyset$. Then the procedure branches over all $g \in T[c]$ and makes a recursive call on each branch with f extended to f' and C reduced by c. Since $f'(v) > f(v)$ for at least one $v \in b$, these calls can happen at most $\mathbf{tpw} \cdot k_\infty$ times until a base case (Line 16) is reached. Thus, the total time spent on `distribute` calls in this case for one bag b is $O(x^{k_\infty \cdot \mathbf{tpw}} \cdot |\mathcal{G}|)$.

Note that the `distribute` procedure essentially forms a bounded search tree algorithm inside the larger algorithm. □

Lemma 3.15: Algorithm 3.3 runs in time $O\left(\left(8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2\right)^{(k_\infty^2+k_\infty)\mathbf{tpw}^2} \cdot |\mathcal{G}| \cdot |V|\right)$, if the tree partition D has width $\mathbf{tpw}$.

Proof. The set $\mathcal{T}$ in Line 7 has a size of at most $2k_\infty \cdot (\ell+1) \cdot (2\mathbf{tpw})^2 \leq 8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2$ (see Observation 3.11). Accordingly, there are at most $1 + \sum_{i \in [k_\infty]} \binom{8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2}{i} \leq 1 + k_\infty \cdot \left(8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2\right)^{k_\infty} \leq \left(8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2\right)^{k_\infty+1}$ subsets of $\mathcal{T}$ to which each individual $v \in b$ can be mapped in Line 8. Consequently, the total number of functions tried out and potentially saved for each bag b is at most $\left(8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2\right)^{(k_\infty+1)\mathbf{tpw}}$.

With each of these functions the `distribute` procedure is called by `process_bag`. Using Lemma 3.14 with $x = \left(8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2\right)^{(k_\infty+1)\mathbf{tpw}}$ reveals that each `process_bag` step on bag b with set of children C in D takes total time at most $O\left(\left(8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2\right)^{(k_\infty^2+k_\infty)\mathbf{tpw}^2} \cdot |\mathcal{G}| \cdot (|C|+1)\right)$.

As `process_bag` is called once with each bag b in D, the total running time of the algorithm is in $O\left(\left(8k_\infty \cdot (\ell+1) \cdot \mathbf{tpw}^2\right)^{(k_\infty^2+k_\infty)\mathbf{tpw}^2} \cdot |\mathcal{G}| \cdot |V|\right)$. □

We are now set to prove Theorem 3.12.

Proof of Theorem 3.12. If we choose b to be the root bag r, then Lemma 3.13 shows that Algorithm 3.3 saves a function in the table entry $T[b]$ if and only if the input instance is YES. Clearly, the `main` procedure of the algorithm then and only then returns YES, accordingly. The specified running time bound of the algorithm is proven by adapting Lemma 3.15. □

Algorithm 3.3: FPT w.r.t. **tpw** $+ k_\infty + \ell$ for MINTIMELINE$_\infty$ (Part 3)

```
   /* Modifies $f$ to cover additional given layers.                    */
31 procedure extend ($S \subseteq V, f : S \to \mathcal{P}([\tau]), R \subseteq S \times [\tau]$)
32     $f' \leftarrow f$
33     while $R \neq \emptyset$ do
34         let $(v, t) \in R$ such that $t$ is minimal
35         $f'(v) \leftarrow f'(v) \cup \{t\}$   // Updates function entry of $v$
36         $R \leftarrow R \setminus \{(v, t') \mid t \leq t' \leq t + \ell\}$
37     end while
38     return $f'$
39 end procedure
   /* Given two vertex sets $S$ and $Q$ and respective vertex
      occurrences for those sets, check_covering validates if all
      temporal edges with endpoints both in $S$ and in $Q$ are
      correctly covered by activating their vertices at the given
      time steps.  If yes, it returns $\emptyset$.  Otherwise, it returns
      the additional vertex occurrences which are needed to cover
      those temporal edges with vertices in $S$.                          */
40 procedure check_covering ($S \subseteq V, f : S \to \mathcal{P}([\tau]), Q \subseteq V,$
    $g : Q \to \mathcal{P}([\tau])$)
41     $R \leftarrow \emptyset$
42     foreach temporal edge $(\{v, w\}, t)$ in $\mathcal{G}$ with $v \in S$ and $w \in Q$ do
43         if $f(v) \cup g(w)$ contains no $t'$ such that $t' \leq t \leq t' + \ell$ then
44             $R \leftarrow R \cup \{(v, t)\}$
45         end if
46     end foreach
47     return $R$
48 end procedure
```

3.3.2 MinTimeline$_+$ Algorithm

Having just described this section's algorithm for deciding MinTimeline$_\infty$, we adapt it for solving MinTimeline$_+$ next (resulting in Algorithm 3.4). Again, we have a useful observation about that problem variant to work with.

Observation 3.16: If the number of temporal edges between two distinct vertices a and b exceeds $k_a + k_b + \ell$ in an instance of MinTimeline$_+$, then it is a No-instance.

Proof. In MinTimeline$_+$, each activation of a or b which lasts for more than one layer induces a cost respective to its interval length and the sum of all those costs is bounded by ℓ. Since a and b can be activated for at most k_a and k_b times respectively, this limits the total number of coverable temporal edges between a and b by $k_a + k_b + \ell$. □

Due to Observation 3.16 we from now on assume to be given instances of MinTimeline$_+$ where each edge is present in at most $2k_\infty + \ell$ layers.

While keeping this observation in mind, we focus on another theorem.

Theorem 3.17: MinTimeline$_+$ in FPT w.r.t. $\mathbf{tpw} + k_\infty + \ell$

MinTimeline$_+$ can be solved in time $2^{O\left(\mathbf{tpw}^4 \cdot (\ell+1)^2 \cdot k_\infty^2\right)} \cdot |V| \cdot |\mathcal{G}|$ when given a tree partition of $\mathcal{G}_\downarrow$ of width $\mathbf{tpw}$.

Similar to Theorem 3.12 earlier, this theorem implies that MinTimeline$_+$ is in FPT w.r.t. $\mathbf{tpw} + k_\infty + \ell$ if combined with the polynomial-time approximation of a minimum-width tree partition described by Bodlaender, Groenland, and Jacob (2022).

The algorithm we present to show Theorem 3.17 is Algorithm 3.4. It adapts the previous Algorithm 3.3 to solve the MinTimeline$_+$ variant and thus features a similar strategy, involving bounded search trees and dynamic programming on an underlying tree partition. The main difference to Algorithm 3.3 is the introduction of saved counters for the sums of interval lengths (called "interval prices") at each saved local solution, in a way that matches the previous Algorithm 3.2. These counters, which are saved in the dynamic programming table, again ensure that no saved solution violates the total interval length limit of MinTimeline$_+$.

Algorithm 3.4: FPT w.r.t. **tpw** $+ k_\infty + \ell$ for MinTimeline$_+$ (Part 1)

Data: A temporal graph $\mathcal{G} = (V, E_1, \ldots, E_\tau)$, a number $\ell \in \mathbb{N}$, an integer vector $k \in \mathbb{N}^{|V|}$, and a tree partition D of $\mathcal{G}_\downarrow$ with a set of bags B. We assume D to be rooted in a bag $r \in B$, meaning that each bag other than r has a unique parent bag.

```
/* Main routine: Dynamic program as in Algorithm 3.3.                        */
procedure main ()
    Initialize globally-accessible array $T$ with $T[b] = \emptyset$ for each $b \in B$
    foreach bag $b \in B$ in bottom-up-order do process_bag($b$)
    if $T[r]$ is $\emptyset$ then return NO else return YES
end procedure
/* Similar to the corresponding procedure of Algorithm 3.3,
   additionaly storing the computed interval price in $T$.                    */
procedure process_bag ($b \in B$)
    $\mathcal{T} \leftarrow \{t \mid (\{a,b\},t) \in \mathcal{G}, \{a,b\} \subseteq (b \cup \text{parent}(b))\}$
    foreach function $f : b \to \mathcal{P}(\mathcal{T})$ with $\sum_{v \in b} |f(v)| \leq \sum_{v \in b} k_v + \ell$ do
        if check_covering($b, f, b, f$) $\neq \emptyset$ then continue
        $x \leftarrow$ distribute($b, f$, children($b$), $\ell$)
        if $x \leq \ell$ then $T[b] \leftarrow T[b] \cup \{(f, x)\}$
    end foreach
end procedure
/* Similar to the respective procedure in Algorithm 3.3, except
   it also returns the minimum interval price for realizing the
   given input.                                                              */
procedure distribute ($b \in B, f : b \to \mathcal{P}([\tau]), C \subseteq B, y \in \mathbb{Z}$)
    if $y < 0$ or $\sum_{v \in b} |f(v)| \leq \sum_{v \in b} k_v + y$ then return $\ell + 1$
    if $C = \emptyset$ then return compute_sum($b, f$)
    choose any $c \in C$
    foreach $(g, x) \in T[c]$ do
        if $x = 0$ and check_covering ($b, f, c, g$) $= \emptyset$ then
            return distribute ($b, f, C \setminus \{c\}$)
        end if
    end foreach
    result $\leftarrow \ell + 1$
    foreach $(g, x) \in T[c]$ do
        $f' \leftarrow$ extend ($b$, $f$, check_covering ($b, f, c, g$))
        result $\leftarrow$ min(result, distribute ($b, f', C \setminus \{c\}, y - x$) $+ x$)
    end foreach
    return result
end procedure
```

Algorithm 3.4: FPT w.r.t. **tpw** $+ k_\infty + \ell$ for MINTIMELINE$_+$ (Part 2)

```
/* A greedy sub-algorithm to calculate the necessary interval
   price for realizing given vertex occurrences of a set.
   Identical to the corresponding procedure of Algorithm 3.2.
   */
procedure compute_sum ($S \subseteq V, f : S \to \mathcal{P}([\tau])$)
    result $\leftarrow 0$
    foreach $v \in S$ do
        if $k_v$ is zero and $f(v)$ is not $\emptyset$ then return $\ell + 1$
        $\mathcal{T} \leftarrow f(v)$
        while $|\mathcal{T}| > k_v$ do
            let $t' \in f(v)$ and $t \in \mathcal{T}$ such that $t' < t$ and $t - t'$ is minimal
            result $\leftarrow$ result $+ (t - t')$
            $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$
        end while
    end foreach
    return result
end procedure

/* Modifies $f$ to cover additional given layers.  Identical to
   the corresponding procedure of Algorithm 3.3.              */
procedure extend ($S \subseteq V, f : S \to \mathcal{P}([\tau]), R \subseteq S \times [\tau]$)
    $f' \leftarrow f$
    while $R \neq \emptyset$ do
        let $(v, t) \in R$ such that $t$ is minimal
        $f'(v) \leftarrow f'(v) \cup \{t\}$   // Updates function entry of $v$
        $R \leftarrow R \setminus \{(v, t') \mid t \leq t' \leq t + \ell\}$
    end while
    return $f'$
end procedure

/* Same as the respective procedure of Algorithm 3.3, except it
   does not need to account for vertex intervals.             */
procedure check_covering ($S \subseteq V, f : S \to \mathcal{P}([\tau]), Q \subseteq V,$
  $g : Q \to \mathcal{P}([\tau])$)
    $R \leftarrow \emptyset$
    foreach temporal edge $(\{v, w\}, t)$ in $\mathcal{G}$ with $v \in S$ and $w \in Q$ do
        if $t \notin f(v) \cup g(w)$ then  $R \leftarrow R \cup \{(v, t)\}$
    end foreach
    return $R$
end procedure
```

For eventually proving Theorem 3.17, we once again highlight important aspects about Algorithm 3.4 by different lemmas.

Lemma 3.18: Let $\mathcal{G}_b$ be the subgraph of $\mathcal{G}$ induced by vertices in bag b and its descendant bags in the tree partition D. Let k' be the corresponding subvector of k, i.e. the unique vector with only $k'_v = k_v$ for each $v \in \mathcal{G}_b$. Let p be the parent bag of b in D, or $p = \emptyset$ if b is the root. Let $\mathcal{T} \subseteq [\tau]$ be the time steps where temporal edges between vertices in $b \cup p$ are present. The procedure `process_bag(b)` of Algorithm 3.4 saves every function $f : b \to \mathcal{P}(\mathcal{T})$ in $T[b]$, for which there is a solution to the instance $(\mathcal{G}_b, k', \ell)$ where each $v \in b$ is activated at time steps $F_v \supseteq f(v)$.

Proof. The proof is nearly identical to that of Lemma 3.13. The only notable difference is the computation of an interval price of each function which is not allowed to exceed ℓ, as we described in the proof of Lemma 3.6.

In this algorithm, the `distribute` procedure therefore returns a number of the minimum interval price with which the function f has a respective solution to the current sub-instance. To this end, it is sufficient to call `compute_sum` (see Lemma 3.5) at each base case of the recursion with the fully extended function and then subsequently add all interval prices of used functions of child bags, since D is a tree partition and thus $b \cap c$ for each two bags b and c in D. Obviously, if there are multiple saved functions for a particular child bag which can be used together with f so that all necessary temporal edges are covered, then `distribute` chooses the option that induces the lowest possible interval price. □

Lemma 3.19: The procedure `distribute`(b, f, C) (see Line 14) runs in time $O(x^{k_\infty \cdot \mathbf{tpw} + \ell} \cdot (|C| + 1) \cdot |\mathcal{G}|)$, where $x = \max_{c \in C} |T[c]|$, whenever called by the `process_bag` procedure.

Proof. The proof is identical to the proof of Lemma 3.14, except that the recursion depth of its second case is bounded by $k_\infty \cdot \mathbf{tpw} + \ell$ instead of $k_\infty \cdot \mathbf{tpw}$. This is due to the fact that with each recursive call either f is extended to f' with $|f'(v)| > |f(v)|$ for at least one $v \in b$ or the parameter y is reduced by at least one. Since $y \leq \ell$, it then takes at most $k_\infty \cdot \mathbf{tpw} + \ell$ recursive calls until the base case in Line 15 is reached. This results in the above total running time of the `distribute` procedure, when called by `process_bag`. □

Lemma 3.20: Algorithm 3.4 runs in time $O\left(\left(2^{(8k_\infty+4\ell)\cdot\mathbf{tpw}^3\cdot(k_\infty\cdot\mathbf{tpw}+\ell)}\right)\cdot|V|\cdot|\mathcal{G}|\right)$, if the tree partition D has width **tpw**.

Proof. The set $\mathcal{T}$ in Line 7 contains at most $(2k_\infty+\ell)\cdot(2\mathbf{tpw})^2 = (8k_\infty+4\ell)\cdot\mathbf{tpw}^2$ elements (see Observation 3.16). Accordingly, there are at most $\left(2^{(8k_\infty+4\ell)\cdot\mathbf{tpw}^2}\right)^{\mathbf{tpw}} = 2^{(8k_\infty+4\ell)\cdot\mathbf{tpw}^3}$ functions which are tried out for each processed bag b in Line 8.

For each of these functions the `distribute` procedure is called by `process_bag`. Using Lemma 3.19 with $x = 2^{(8k_\infty+4\ell)\cdot\mathbf{tpw}^3}$ reveals that each `process_bag` step on bag b with set of children C in D takes total time at most $O\left(\left(2^{(8k_\infty+4\ell)\cdot\mathbf{tpw}^3}\right)^{k_\infty\cdot\mathbf{tpw}+\ell}\cdot(|C|+1)\cdot|\mathcal{G}|\right) = O\left(2^{(8k_\infty+4\ell)\cdot\mathbf{tpw}^3\cdot(k_\infty\cdot\mathbf{tpw}+\ell)}\cdot(|C|+1)\cdot|\mathcal{G}|\right)$,

As `process_bag` is called once with each bag b in D, the overall running time of the algorithm is in $O\left(2^{(8k_\infty+4\ell)\cdot\mathbf{tpw}^3\cdot(k_\infty\cdot\mathbf{tpw}+\ell)}\cdot|V|\cdot|\mathcal{G}|\right)$. □

Similar to the way we showed Theorem 3.12 previously, we can use Algorithm 3.4 and its corresponding lemmas to prove Theorem 3.17.

Proof of Theorem 3.17. If we choose b to be the root bag r, then Lemma 3.18 shows that Algorithm 3.4 saves a function in the table entry $T[b]$ if and only if the input instance is YES. Clearly, the `main` procedure of the algorithm then and only then returns YES, accordingly. The specified running time bound of the algorithm is proven by adapting Lemma 3.20. □

3.4 Further Research regarding the Edge-Treewidth Parameter

We previously showed that MINTIMELINE$_\infty$ and MINTIMELINE$_+$ are in FPT w.r.t. the parameter $\mathbf{tpw}+k_\infty+\ell$ (see Theorems 3.12 and 3.17). We now want to extend these results from tree partition width to a parameter recently introduced by Magne et al. (2023), namely *edge-treewidth* (from now on abbreviated **etw**). So far, seemingly, no relationship between the parameters **tpw** and **etw** was known. But studying these two graph parameters, we found out that **tpw** is always polynomially upper-bounded by **etw**—implying that our FPT-algorithm w.r.t. $\mathbf{tpw}+k_\infty+\ell$ is indeed an FPT-algorithm w.r.t. $\mathbf{etw}+k_\infty+\ell$ as well.

Theorem 3.21: Edge-Treewidth Quadratically Upper-Bounds Tree Partition Width

For any graph G, $\mathbf{tpw}(G) \leq 12 \cdot \mathbf{etw}(G)^2$.

In order to prove this theorem, we make use of the fact that **etw** is related to the treewidth and the maximum degree of the blocks (i.e. biconnected components) of G. We will use the following theorem by Ding and Oporowski (1995) on each block in G. This way, we get an appropriate tree partition for each block, which we can later piece together to obtain a tree partition of appropriate width for G.

Theorem 3.22: Ding and Oporowski (1995), restated

For any $G = (V, E)$ and any set of vertices $S \subseteq V$ with $4 \cdot \mathbf{tw}(G) \leq |S| \leq 12 \cdot \mathbf{tw}(G)\Delta(G)$, there is a tree partition of G with width at most $24 \cdot \mathbf{tw}(G)\Delta(G)$ where all vertices from S are in the same bag.

Proof to Theorem 3.21. Let B be the set of all blocks and C be the set of all cut vertices of G. Let T be a block-rooted block-cut tree of G, i.e. $T = (B \cup C, E', r)$ is a graph with a vertex set $B \cup C$, an edge set E' and a dedicated root node $r \in B$, such that $\{x, y\} \in E'$ if and only if $x \in C \cap y$. This T is a tree, because otherwise two distinct blocks were connected via multiple disjoint paths in G, which would contradict them being biconnected components.

For every block $b \in B \setminus \{r\}$, let $s_b \in C$ be the parent vertex of b in T (which is a cut vertex in G). Let $b' = b \setminus \{s_b\}$. We next show that $\mathbf{tpw}(b') \leq 24 \cdot \mathbf{tw}(b)\Delta(b)$. If b has fewer than $4 \cdot \mathbf{tw}(b) + \Delta(b)$ vertices, this is trivially attainable by putting all vertices in b' in one bag. Otherwise, let X be a set of vertices in b with $N_b[s_b] \subseteq X$ and $|X| = 4 \cdot \mathbf{tw}(b) + \Delta(b)$. Using Theorem 3.22 on the graph b and the vertex set X, we obtain that there is a tree partition of b with a width of at most $24 \cdot \mathbf{tw}(b)\Delta(b)$ and where crucially all vertices of $N_b[s_b]$ are in the same bag. Since $b' = b \setminus \{s_b\}$, the same tree partition is also valid for b', after removing s_b from that bag.

Clearly, the subgraph r of G also has a tree partition width of at most $24 \cdot \mathbf{tw}(r)\Delta(r)$ using a similar argument involving Theorem 3.22. Putting together all those tree partitions of each b' and of r, we can now construct a tree partition of G with width at most $24 \cdot \max_{b \in B} (\mathbf{tw}(b) \cdot \Delta(b))$. For this we only have to link one bag of the previously regarded tree partition of each b' — namely the bag containing the neighbors of s_b in b—to the one bag containing s_b (which is part of a different block's tree partition, namely the parent block of s_b in T). Note that since T is a block-cut tree and we put each s_b in exactly one bag, which is in turn adjacent to all other bags including neighbors of s_b, this scheme indeed yields a tree partition for G.

Magne et al. (2023) already proved that $\mathbf{tw}(b) \leq \mathbf{etw}(G)$ and $\Delta(b) \leq \frac{\mathbf{etw}(G)}{2}$ for each $b \in B$. As we just showed that $\mathbf{tpw}(G) \leq 24 \cdot \max_{b \in B} (\mathbf{tw}(b) \cdot \Delta(b))$, this concludes our proof. □

Combining Theorem 3.21 with Theorem 3.12 and Theorem 3.17, the following result about NETWORK UNTANGLING becomes apparent.

Corollary 3.23: Network Untangling in FPT w.r.t. $\mathbf{etw} + k_\infty + \ell$

MINTIMELINE$_\infty$ and MINTIMELINE$_+$ are both fixed-parameter tractable w.r.t. $\mathbf{etw} + k_\infty + \ell$.

Theorem 3.21 gives a true upper bound in the sense that $\mathbf{etw}(G)$ is not upper-bounded by $\mathbf{tpw}(G)$ as well. To corroborate this claim, consider the graph $K_{2,n}$ for any integer $n \geq 2$. It is clear that this graph has a constant tree partition width, since one can just put the two left-sided vertices together in one bag and each other vertex in a separate bag. But since the whole graph consists of only one block and $\Delta(K_{2,n}) = n$, its edge-treewidth is polynomially tied to n as shown by Magne et al. (2023).

We additionally note that the parameters $\mathbf{etw}$ and $\mathbf{vc}$ (vertex cover number) are parametrically incomparable. This further distinguishes $\mathbf{etw}$ from $\mathbf{tpw}$, as $\mathbf{tpw}(G) \leq \mathbf{vc}(G)$ for each simple graph G. The incomparability claim can easily be supported by the example of a $K_{2,n}$ as stated above for one direction and a P_n for the other direction.

4 Notes on the Uniform Versions of Network Untangling

At the beginning of this work, we defined our two main problem variants to be non-uniform. In particular, every vertex $v \in V$ in the input instance has a separate limit k_v of activations, together compactly represented in the vector k. This notion stands in contrast with previous conceptions of the problem, e.g. by Rozenshtein, Tatti, and Gionis (2021) and Froese, Kunz, and Zschoche (2022), which usually perceived NETWORK UNTANGLING to be uniform. In those formulations, one is thus given only a number k instead of a vector and each vertex v is allowed to be activated at most k times in any solution.

In this chapter, we want to explain how our previous results carry over to the uniform problem variants. Clearly, our presented algorithms work for them similarly by just setting each k_v to k in their input. For the hardness results, we next describe how to modify our reductions in order to output equivalent uniform instances. All these modifications work by adding more layers and temporal edges.

4.1 NP-Hardness on Bipartite Graphs with Constant Number of Layers

Among our reductions, the simplest to adjust for UNIFORM MINTIMELINE$_\infty$ and UNIFORM MINTIMELINE$_+$ is Reduction 2.11. Hence, we start with proving the following adaptation of Theorem 2.7:

Corollary 4.1: Uniform NP-Hardness on Bipartite Graphs with $\tau = 7$

UNIFORM MINTIMELINE$_\infty$ and UNIFORM MINTIMELINE$_+$ are NP-hard even if the underlying graph of the input is bipartite, $\ell = 0$, $\tau = 7$ and $\Delta = 12$.

Proof. We use the steps described by Reduction 2.11, but set $k = 2$ uniformly. We additionally introduce a new vertex Y and four new layers $\mathcal{G}_4$, $\mathcal{G}_5$, $\mathcal{G}_6$ and $\mathcal{G}_7$, which each contain exactly one temporal edge between X and Y. Clearly, X and Y then must be activated in two of these additional layers each in order to build up a solution and X can consequently not be active in the first three layers. This makes this uniform instance equivalent to the original constructed instance while increasing τ to seven. It is easy to see that the underlying graph is still bipartite with a maximum degree of twelve. □

4.2 Star NP-Hardness

We also showed that MINTIMELINE$_\infty$ with $\ell = 0$ is NP-hard on stars. By modifying Reduction 2.3 and revisiting Theorem 2.1, we prove another corollary.

Corollary 4.2: Uniform NP-Hardness on Star Graphs

UNIFORM MINTIMELINE$_+$ and UNIFORM MINTIMELINE$_\infty$ are both NP-hard even if the underlying graph is a star, each layer contains at most 3 edges, and $\ell = 0$.

Proof. W.l.o.g. we now assume that each vertex in the input instance of Reduction 2.3 has at least two neighbors, such the original instructions stated by Reduction 2.3 only construct layers with two or more edges. Additionally to following those instructions, we then want to add $k - 1$ separate layers for each vertex $e \in E$ to cover. This way, we can increase each k_e to $k_c = k$ and effectively create a uniform instance. Naturally, the instances have to be equivalent and the constructed temporal graph must still be an underlying star.

Fortunately, we can just add $k - 1$ layers for each $e \in E$ only including the edge $\{e, c\}$ to achieve this. It is never optimal for those additional temporal edges to be covered by the center vertex c, because the total number of temporal edges between c and e becomes $k + 1$ and thus only one of them has to be covered by c (remember that e has no other neighbors than c). This one temporal edge can always be in a layer where activating c covers even more edges, i.e. one of the layers originally created by Reduction 2.3. As a consequence, the remaining proof works as for Theorem 2.1, when only considering such optimal solutions. □

4.3 Caterpillar Tree NP-Hardness

For modifying Reduction 2.5 such that it outputs an equivalent instance of UNIFORM MINTIMELINE$_\infty$ with $\ell = 0$, we combine the approaches used in the proofs of the previous two Corollaries 4.1 and 4.2, yielding the following corollary of Theorem 2.4.

Corollary 4.3: Uniform NP-Hardness on Caterpillar Trees

UNIFORM MINTIMELINE$_\infty$ and UNIFORM MINTIMELINE$_+$ with $\ell = 0$ are both NP-hard even if the underlying graph is a caterpillar tree with a maximum degree of 4.

Proof. Reduction 2.5 introduced three pairwise disjoint sets of vertices: E, E^*, and E^{**}. Remember that we used n to denote the number of input vertices, as well as k to denote the respective input parameter of VERTEX COVER, and we constructed a vector k' for the output instance of MINTIMELINE$_\infty$. In the following we extend that output instance, so that we can set $k'_v = n$ for each included vertex v, while keeping the instances equivalent.

Each $e^* \in E^*$ had $k'_{e^*} = n - k$ and each $e^{**} \in E^{**}$ had $k'_{e^{**}} = k$ previously. For each $e^* \in E^*$ we thus add a unique new vertex with a single edge to e^* in $n + k$ additional unique layers. This new vertex may of course also be activated at exactly n time steps in order to make the instance uniform. Since there are $n + k$ such newly-created layers, e^* has to cover at least k of them. This amounts to $n - k$ original layers (described by Reduction 2.5) which e^* can cover, as desired. We do the same thing for each $e^{**} \in E^{**}$, except that we create $2n - k$ new layers there. This increases the maximum degree of the constructed underlying graph by one in comparison to the original reduction.

For each $e \in E$, we cannot add an additional vertex in the same way, since we still want the constructed underlying graph to be a caterpillar tree. Instead, we add $n - 1$ new layers containing only the edge $\{e, e^{**}\}$. We know that they have to be covered by e, since the budget of activations of e^{**} is already used up on other layers (described by Lemma 2.6 in addition to the layers added above). □

4.4 W[2]-Hardness w.r.t. $k_\infty + \ell$ on Temporal Graphs with dlf $= 1$

The last theorem we want to adapt to the uniform problem variants is Theorem 3.8. In order to do this, we use a similar adaptation strategy as in the proof of Corollary 4.2, arriving at the following:

Corollary 4.4: Uniform W[2]-Hardness w.r.t. dlf $+ k_\infty + \ell$

UNIFORM MINTIMELINE$_+$ and UNIFORM MINTIMELINE$_\infty$ are both W[2]-hard w.r.t. k, even if the underlying graph's distance to a linear forest is one and $\ell = 0$.

Proof. Additionally to the steps described by Reduction 3.10, for each S_j and for each $u \in S_j$, we add $k-2$ unique layers to $\mathcal{G}$ which only include the edge $\{\mathcal{S}_j^u, X\}$. We also add a new vertex for each $\mathcal{S}_j^0$ and for each $\mathcal{S}_j^{m+1}$, both having edges to $\mathcal{S}_j^0$ and $\mathcal{S}_j^{m+1}$ respectively in exactly $2k$ new unique single-edge layers. Afterwards, we set $k'_v = k$ for each vertex v included in $\mathcal{G}$, thereby effectively creating a uniform instance (remember that $k'_X = k$).

Every solution to the input instance is still a solution to that uniform output instance, since if no additionally added layer is covered by X, then the budget of allowed activations for each other vertex in the original layers of Reduction 3.10 effectively remains unchanged (we added exactly as many layers as we can cover by the budget increases to k). Thus, this direction in the proof of Theorem 3.8 still holds.

The other direction also works as in the proof of Theorem 3.8, except the numbers need to be updated. If the input is a NO-instance, there is still at least one $\mathcal{S}_j$ where no $v \in \mathcal{S}_j$ has an incident temporal edge covered by X, since k'_X remained unchanged. The number of temporal edges incident to vertices in this $\mathcal{S}_j$ is then $(|S_j|+1) + |S_j| + |S_j| \cdot (k-2) + 2 \cdot 2k = 4k + k \cdot |S_j| + 1$, each lying in a distinct layer. The newly-created vertices at the start and at the end of the path can together cover $2k$ of those temporal edges, such that $2k + k \cdot |S_j| + 1 = k \cdot (|S_j| + 2) + 1$ temporal edges remain to be covered with the $|S_j| + 2$ vertices in $\mathcal{S}_j$. Since each of them can be activated at a number of k time steps at most, we have a NO-instance, as desired.

Clearly, the constructed graph has still only a distance of one to a linear forest, thus concluding the proof. □

5 Conclusion and Outlook

We analyzed two variants of Network Untangling in the context of computational complexity, focussing on restricted and parameterized settings. In particular, we combined the study regarding structural graph parameters with studying parameters specific to Network Untangling. By doing so, we found several new hardness results but could also formulate multiple parameterized exact algorithms.

In the next Section 5.1, we further review our findings. Subsequently, in Section 5.2, we conclude by discussing opportunities for future research based on our discoveries, including potential enhancements to our algorithms.

5.1 Summary

We provided an extensive study on the parameterized complexity landscape of Network Untangling regarding any combination of the graph parameters **tw**, **fvs**, **dlf**, **tpw**, **etw**, and **vc** on one side and problem parameters specific to Network Untangling and temporal graphs, namely ℓ, $k_\infty + \ell$, and τ, on the other side (see Table 1.1 at the beginning of this work). To the best of our knowledge, we are the first to explicitly research a parameterization of MinTimeline$_\infty$ and MinTimeline$_+$ with graph parameters that are smaller than the number of input vertices. Thus, this work helps demarcating the border of fixed-parameter tractability especially in structurally restricted settings. We mainly focussed on the non-uniform formulations of the problems, but our presented results carry over to uniform settings as well (Chapter 4).

Our two main results establish that both problem variants are fixed-parameter tractable when parameterized by either $\mathbf{tw} + \tau$ or $\mathbf{tpw} + k_\infty + \ell$ (Theorems 3.1, 3.4, 3.12 and 3.17). We provided concrete algorithm descriptions in both cases, employing the strategy of dynamic programming in the former and a fusion of dynamic programming and bounded search trees in the latter case. We proved that FPT-algorithms w.r.t. $\mathbf{tw} + k_\infty + \ell$, which is smaller than $\mathbf{tw} + \tau$ or $\mathbf{tpw} + k_\infty + \ell$ at least for the MinTimeline$_\infty$ variant, cannot exist unless FPT $=$ W[2] (Theorem 3.8). However, we could ascertain that Network Untangling is in XP regarding this usually smaller parameter.

Additionally, we found three more graph structural cases where both problems remain NP-hard, thereby adding to previous knowledge. In particular, we strengthened a result of Froese, Kunz, and Zschoche (2022) by essentially proving that it also holds on bipartite temporal graphs (Theorem 2.7). We also showed that MINTIMELINE$_+$ and MINTIMELINE$_\infty$ are NP-hard even if $\mathbf{vc} = 1$ and $\ell = 0$ (Theorem 2.1). In settings with constant ℓ, this displays a clear tractability border between **vc** and the number of vertices n, considering that Froese, Kunz, and Zschoche (2022) proved MINTIMELINE$_+$ to be in FPT and MINTIMELINE$_\infty$ to be in XP for $n + \ell$. We further observe that we could not distinguish the parameterized complexity of MINTIMELINE$_+$ and MINTIMELINE$_\infty$ in any of our settings, despite the fact that Froese, Kunz, and Zschoche (2022) found a case where the former is in FPT and the latter is W[1]-hard.

Last but not least, we considered edge-treewidth, a graph parameter recently introduced by Magne et al. (2023). We then proved it to be related to tree partition width in the sense that the edge-treewidth of a graph always polynomially upper-bounds its tree partition width.

5.2 Future Research Opportunities

At their current state, our presented algorithms are primarily suited for classification purposes. To make them more useful in practical applications, several enhancements can be considered: First, it would certainly be beneficial to refine the analysis of which vertex occurences an algorithm needs to try out. Additionally, one could quickly discard branches that are recognized as not leading to an improved solution (i.e. by establishing powerful branch and bound methods) and utilize efficient data structures, among other ideas. In practice, it may also be advantageous to integrate our algorithms with heuristic approaches. For instance, strategies could involve initially decomposing a temporal graph's dense areas using a heuristic, followed by the utilization of our exact algorithms to provide solutions of the resulting subgraphs.

On the theoretical front, in addition to refining our algorithms, we find the following open questions to be particularly interesting for expanding our research on NETWORK UNTANGLING:

- Do MINTIMELINE$_\infty$ and/or MINTIMELINE$_+$ admit polynomial (Turing) kernels concerning any parameter for which we showed them to be fixed-parameter tractable, such as $\mathbf{vc} + \tau$?
- Are MINTIMELINE$_\infty$ and MINTIMELINE$_+$ in FPT if parameterized by the combination of the parameters *treedepth*, k_∞ and ℓ?
- What about the parameter $\mathbf{tpw} + k_\infty$ in the setting where ℓ is large? Our presented FPT-algorithms only work regarding $\mathbf{tpw} + k_\infty + \ell$, since they rely on bounding the number of time steps at which any specific edge is present.
- What running time lower bounds (e.g. based on the *Exponential Time Hypothesis*) exist for these problems regarding the cases where they exhibit fixed-parameter tractability?

Of course, exploring alternative—potentially further restricted—versions of NETWORK UNTANGLING may also unveil novel research-worthy questions.

References

Akrida, Eleni C., George B. Mertzios, Paul G. Spirakis, and Viktor Zamaraev. 2020. *Temporal vertex cover with a sliding time window. Journal of Computer and System Sciences* 107 (February 1, 2020). (Cited on pages 13, 19)

Alman, Josh, Matthias Mnich, and Virginia Vassilevska Williams. 2020. *Dynamic Parameterized Problems and Algorithms. ACM Transactions on Algorithms* 16, no. 4 (July 6, 2020). (Cited on page 13)

Biró, M., M. Hujter, and Zs. Tuza. 1992. *Precoloring extension. I. Interval graphs. Discrete Mathematics* 100, no. 1 (May 15, 1992). (Cited on page 23)

Bodlaender, Hans L. 1996. *A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. SIAM Journal on Computing* 25 (6). (Cited on pages 30, 31, 35)

Bodlaender, Hans L., Carla Groenland, and Hugo Jacob. 2022. *On the Parameterized Complexity of Computing Tree-Partitions.* 17th International Symposium on Parameterized and Exact Computation (IPEC 2022). Schloss-Dagstuhl - Leibniz Zentrum für Informatik. (Cited on pages 39, 45)

Bodlaender, Hans L., Klaus Jansen, and Gerhard J. Woeginger. 1994. *Scheduling with incompatible jobs. Discrete Applied Mathematics* 55, no. 3 (December 13, 1994). (Cited on pages 23, 24)

Ding, Guoli, and Bogdan Oporowski. 1995. *Some results on tree decomposition of graphs. Journal of Graph Theory* 20 (4). (Cited on page 50)

Dondi, Riccardo. 2022. *Insights into the Complexity of Disentangling Temporal Graphs,* vol. 3284. Proceedings of the 23rd Italian Conference on Theoretical Computer Science (ICTCS 2022). September 7, 2022. (Cited on pages 9, 13)

Dondi, Riccardo. 2023. *Untangling temporal graphs of bounded degree. Theoretical Computer Science* 969 (August 21, 2023). (Cited on pages 9, 13)

Dondi, Riccardo, and Manuel Lafond. 2023. *An FTP Algorithm for Temporal Graph Untangling* (July 3, 2023). (Cited on pages 8, 9, 13)

Dondi, Riccardo, and Alexandru Popa. 2023. *Timeline Cover in Temporal Graphs: Exact and Approximation Algorithms.* In *Combinatorial Algorithms.* Lecture Notes in Computer Science. Cham: Springer Nature Switzerland. (Cited on pages 9, 13)

Downey, R. G., and M. R. Fellows. 1999. *Parameterized Complexity.* Monographs in Computer Science. New York, NY: Springer. (Cited on page 36)

Fluschnik, Till, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. 2020a. *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. Springer International Publishing. (Cited on page 18)

Fluschnik, Till, Hendrik Molter, Rolf Niedermeier, Malte Renken, and Philipp Zschoche. 2020b. *Temporal graph classes: A view through temporal separators. Theoretical Computer Science* 806 (February 2, 2020). (Cited on page 8)

Fluschnik, Till, Rolf Niedermeier, Valentin Rohm, and Philipp Zschoche. 2022. *Multistage Vertex Cover. Theory of Computing Systems* 66, no. 2 (April 1, 2022). (Cited on page 13)

Fluschnik, Till, Rolf Niedermeier, Carsten Schubert, and Philipp Zschoche. 2023. *Multistage s–t Path: Confronting Similarity with Dissimilarity. Algorithmica* 85, no. 7 (July 1, 2023). (Cited on page 18)

Froese, Vincent, Pascal Kunz, and Philipp Zschoche. 2022. *Disentangling the Computational Complexity of Network Untangling* (April 6, 2022). (Cited on pages 4, 5, 8, 9, 13, 22, 23, 52, 56, 57)

Garey, M. R., D. S. Johnson, and L. Stockmeyer. 1976. *Some simplified NP-complete graph problems. Theoretical Computer Science* 1, no. 3 (February 1, 1976). (Cited on pages 19, 23, 24)

Holme, Petter. 2015. *Modern Temporal Network Theory: A Colloquium. The European Physical Journal B* 88, no. 9 (September 21, 2015). (Cited on page 8)

Holme, Petter, and Jari Saramäki. 2012. *Temporal networks. Physics Reports,* Temporal Networks, 519, no. 3 (October 1, 2012). (Cited on page 8)

Iwata, Yoichi, and Keigo Oka. 2014. *Fast Dynamic Graph Algorithms for Parameterized Problems.* In *Algorithm Theory – SWAT 2014.* Springer International Publishing. (Cited on page 13)

Magne, Loïc, Christophe Paul, Abhijat Sharma, and Dimitrios M. Thilikos. 2023. *Edge-treewidth: Algorithmic and combinatorial properties. Discrete Applied Mathematics* 341 (December 31, 2023). (Cited on pages 11, 49, 51, 57)

Molter, Hendrik, Malte Renken, and Philipp Zschoche. 2021. *Temporal Reachability Minimization: Delaying vs. Deleting.* In *46th International Symposium on Mathematical Foundations of Computer Science (MFCS 2021),* vol. 202. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik. (Cited on page 8)

Rozenshtein, Polina, Nikolaj Tatti, and Aristides Gionis. 2021. *The Network-Untangling Problem: From Interactions to Activity Timelines. Data Mining and Knowledge Discovery* 35, no. 1 (January 1, 2021). (Cited on pages 8–10, 13, 52)

Wang, Yishu, Ye Yuan, Yuliang Ma, and Guoren Wang. 2019. *Time-Dependent Graphs: Definitions, Applications, and Algorithms. Data Science and Engineering* 4, no. 4 (December 1, 2019). (Cited on page 8)