Utrecht University

Faculty of Science

# On the Fixed Parameter Tractability of Minimum Temporal Connectivity

Master Thesis

*Jens Heuseveldt*

Computing Science

*Supervisors*:

Prof. dr. Hans Bodlaender

dr. Johan van Rooij

November 14, 2022

## Abstract

Temporal graphs allow encoding time-dependent information in graphs. However, problems on temporal graphs tend to be more complex than their counterparts on static graphs. The Minimum Temporal Connectivity and Maximum Temporal Matching problems are both NP-hard. In this thesis we provide an algorithm that proves that the Minimum Temporal Connectivity problem is fixed parameter tractable, using the graph lifetime and treewidth as combined parameter.

# 1 Overview and introduction

Many real-world problems, like assigning tasks to people, finding your way home or building an energy network, can be modelled as graph problems. Graphs, however, are static, and it can be difficult to incorporate a time-element into the model. One possible solution to this is to use temporal graphs. As all edges of a temporal graph have a timestamp, we can encode time-dependent information in it. This can include roads being built or demolished, a flow through a graph that can wait at locations during travel, or evolving social networks.

There exist many problems on static graphs where the temporal counterparts appear to be more complex. We will study the (maximum) temporal matching (TM) and minimum temporal connectivity (MTC) problems, which are the temporal counterparts of the maximum matching and minimum spanning tree problems respectively.

Where a path from one vertex to another implies a path the other way around in normal, undirected graphs, this is not the case for temporal graphs. The MTC problem is therefore split in two general variants, one requires only one vertex to be connected to all others, where the other variant requires all pairs of vertices to be mutually connected. We denote the single-source variant as $r$-MTC and the all-pairs variant as MTC.

Both these problems are NP-hard. Unless $P = NP$, we need to put extra constraints on the problem in order to solve it in polynomial time. One technique to do that is *fixed parameter tractability*[7], FPT for short. We also denote the class of problems that are fixed parameter tractable as FPT. For a problem to be FPT with respect to a parameter, there must be an algorithm that can solve instances of the problem in polynomial time, given that the chosen parameter remains constant and the polynomial is independent of the parameter.

Axiotis et al.[1] have shown that the MTC problem on a tree is in FPT when parametrized by only the lifetime of the graph, usually denoted by $\tau$, and it is 2-approximable on a cycle. They also proved that the general MTC problem is APX-hard. Additionally, they gave us an FPT algorithm for $r$-MTC, parametrized by treewidth+$\tau$, a polynomial time algorithm for $r$-MTC on unweighted graphs and have shown that the MTC problem is APX-hard, even in the unweighted case. For the general $r$-MTC problem, Kempe et al.[3] proved that the underlying graph of a solution always is a tree.

For temporal matching, we require the solution to have as many (timestamped) edges as possible. No vertex may be incident to a selected edge twice within a given cooldown period.

Molter[6] worked on many temporal graph problems and proved that TM is NP-hard. Mertzios et al.[5] extended this result to graphs with limited cooldown and lifetime. In particular, TM is NP-hard even with a cooldown of 2 and a lifetime of 3, or with a path as underlying graph with a cooldown of 2. The former result also holds when the underlying graph is complete and the graph has a lifetime of 5.

An FPT algorithm parametrized by the cooldown and maximum matching size of the underlying graph was also given by Mertzios et al[5]. Later Zschoche[8] improved that algorithm to be exponentially faster in terms of the cooldown peroid.

In this thesis I present an algorithm to solve MTC in linear time for graphs of bounded treewidth and lifetime. For that, we first extend the notion of a nice tree decomposition and introduce the Activation Propagation problem, which we will use as subproblem in the algorithm for Minimum Temporal Connectivity.

## 2 Preliminaries

A **graph** is a tuple $G = (V_G, E_G)$ with $V_G$ the set of **vertices** and $E_G$ the set of **edges**. Each edge is an unordered pair of two vertices of $V_G$. The subscript $_G$ may be omitted if the corresponding graph is clear from the context. Unless otherwise specified, $n$ is the number of vertices $|V|$ and $m$ the number of edges $|E|$.

A **temporal graph** is a tuple $G = (V_G, E_G, \tau_G)$. Again, we will omit the subscript $_G$ if that does not introduce ambiguity. All edges $e_i \in E$ are a tuple $((v_{i,1}, v_{i,2}), t_i)$ with $v_{i,1}, v_{i,2} \in V$ and $1 \leq t_i \leq \tau$, meaning that vertices $v_{i,1}$ and $v_{i,2}$ are **adjacent** at time $t_i$. Unless stated otherwise, edges are undirected, which means that we consider $((v_1, v_2), t)$ and $((v_2, v_1), t)$ as equal. Figure 1 shows an example of a temporal graph.

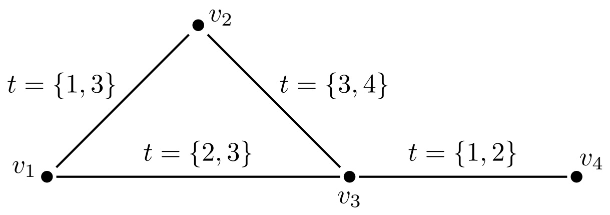

Figure 1: Example of a temporal graph.

The weight (or cost) of an edge $e$ is denoted as $w(e)$ (or $c(e)$). For unweighted graphs, $w(e) = 1$ for all edges. The set with all edges at a given time $t$ is denoted by $E_G^t = \{e \mid (e,t) \in E_G\}$. For a temporal graph $G$, we have a **snapshot** $G^t = (V_G, E_G^t)$, as shown in Figure 2. The **underlying graph** $G^\downarrow$ is the graph where all edges have their time parameter removed: $G^\downarrow = (V_G, E_{G^\downarrow})$ where $E_{G^\downarrow} = \bigcup_{1 \leq t \leq \tau} E_G^t$. Note that both a snapshot and the underlying graph are normal graphs, rather than temporal graphs.

The TM problem is defined as follows. Given a temporal graph $G$ and a cooldown period $c$, find a set $M \subseteq E$ of maximum cardinality such that for any two different $(e_1, t_1), (e_2, t_2) \in M$ we either have that $|t_1 - t_2| \geq c$ or $e_1$ and $e_2$ do not share a vertex. An example can be found in Figure 3.

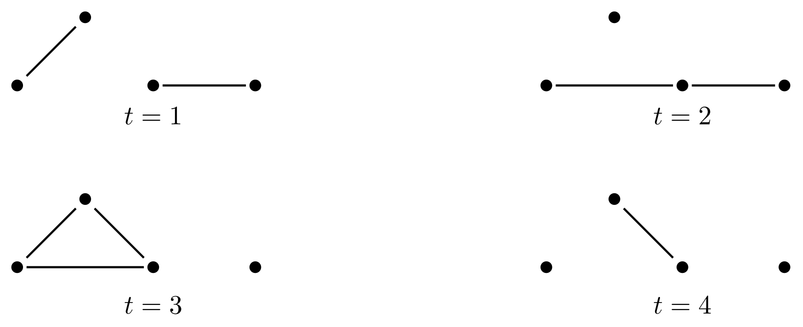

Figure 2: All snapshots of the graph in Figure 1

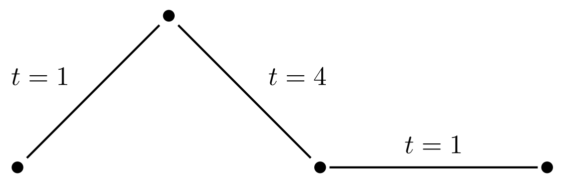

Figure 3: Optimal solution for TM on the graph in Figure 1, for both cooldown period 2 and 3.

A **temporal path** is a series of alternating vertices and timestamps $\{v_1, t_1, v_2, t_2, \ldots, v_k, t_k, v_{k+1}\}$ such that $((v_i, v_{i+1}), t_i) \in E$ for all $1 \leq i \leq k$ and $t_1 \leq t_2 \leq \ldots \leq t_k$. If all timestamps are unique, the temporal path is **strict**. When there exists a (strict) temporal path from vertex $v_1$ to $v_2$, we say that there is a **(strict) temporal connection** from $v_1$ to $v_2$. The vertex $v_2$ is then **temporally connected** to $v_1$. If $v_1$ is connected to $v_2$ and vice versa, $v_1$ and $v_2$ have a **mutual temporal connection**.

The (weighted) $r$-MTC problem is defined as follows. Given a temporal graph $G$ and a vertex $r \in V$, find a set $M \subseteq E$ of minimum cardinality (or total weight) such that every vertex $v \in V$ is temporally connected to $r$ in $(V, M, \tau)$. The MTC problem is defined similarly, but requires all pairs of vertices to be mutually temporally connected. Figure 4 shows an example for both problems.

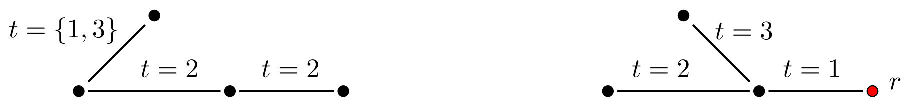

Figure 4: Optimal solutions for MTC (left) and $r$-MTC (right) on the graph in Figure 1.

## 2.1 Tree decompositions

A **tree decomposition** $H$ of a graph $G = (V, E)$ is a graph with a number of useful properties. The graph $H$ is a tree and we will call its vertices **nodes** in order to prevent confusion with the vertices of $G$. Every node $X_i$ of $H$, is a subset of $V$ and every vertex of $G$ occurs in at least one node of $H$. When we refer to $X_i$ specifically for its vertices, we usually refer to it as **bag**. For every edge $(v_1, v_2) \in E$, there is at least one node of $H$ that contains both $v_1$ and $v_2$. Finally, for every $v \in V$, the induced subgraph of $H$ of the nodes that contain $v$ is connected. The width of such a decomposition is $\max_i |X_i| - 1$. The **treewidth** of a graph is the smallest width among all possible tree decompositions.

When we select one of the leaves as root node, we have a **rooted tree decomposition**. A **nice tree decomposition** [4, Definition 13.1.5] is a rooted tree decomposition where each node $X_i$ is one of four types:

- Join node: $X_i$ has two child nodes that both have the same vertex set as $X_i$.
- Introduce node: $X_i$ has one child node $X_j$ where $X_i$ has the same vertex set as $X_j$, but with one vertex added to it ($X_i \supseteq X_j$ and $|X_i| = |X_j| + 1$).
- Forget node: $X_i$ has one child node $X_j$ that has the same vertex set, but with one additional vertex ($X_i \subseteq X_j$ and $|X_i| + 1 = |X_j|$).
- Leaf node: $X_i$ has no child nodes and consists of only one vertex.

Any tree decomposition can be turned into a nice tree decomposition with the same width and at most $4n$ nodes (see [4, Lemma 13.1.3]). In addition to these nodes, we add a new type of node, the **introduce edge** node, similar to the one described by Cygan et al.[2, Definition 2.3]. Every (temporal) edge is associated with exactly one introduce edge node, and at every introduce edge node, exactly one (temporal) edge is introduced. Every introduce edge node has exactly one child node with the same vertex set. Both endpoints of the (temporal) edge associated with the introduce edge node must be in the vertex set of the node. As in [2], we add a special root node which is a forget node and the only node $i$ for which $X_i = \emptyset$.

For every graph we can trivially construct a tree decomposition of width $|V| - 1$: From the single leaf node with an arbitrary vertex, we introduce add all vertices, then all edges and finally forget all vertices one-by-one until we have an empty root node. Multiple nice tree decompositions may exist for a single graph. Figure 5 shows a non-trivial tree decomposition of the graph in Figure 1.

For any given node $X_i$, the vertices in that bag are called the **active vertices**. Vertices for which the corresponding forget node is in the subtree rooted at $i$ are **forgotten vertices**, which, together with the active vertices, form the **discovered vertices**. The undiscovered vertices are precisely the vertices that do not occur in the subtree rooted at $i$. Due to the separator property of tree decompositions, nodes cannot contain both a forgotten and an undiscovered vertex of another node.

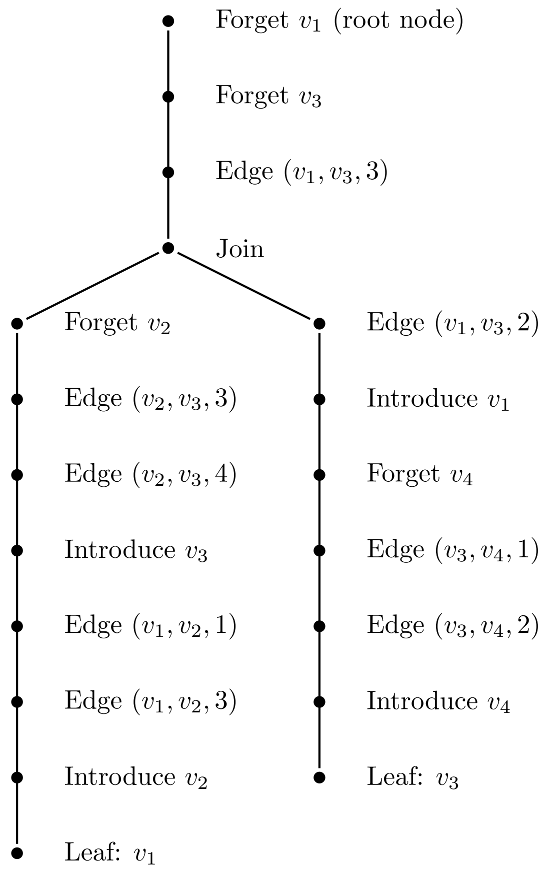

Figure 5: A tree decomposition for the graph in Figure 1.

## 2.2 Activation Propagation

In this section we will define the ACTIVATION PROPAGATION problem, which we will use in our algorithm for MTC. It is defined as follows:

Let $L$ and $R$ be two sets of vertices and $G = (L \cup R, E)$ be a bipartite graph with each edge in $E$ connecting a vertex in $L$ with a vertex in $R$. Both vertex sets are partitioned into groups. Select a number of edges $E' \subseteq E$ such that the following holds: for every possible selection of one vertex in every group in $L$, which we call the activated vertices, there must be one group in $R$ in which all vertices are connected (through $E'$) to an activated vertex in $L$.

Figure 6 shows all possible ways to activate vertices in $L$ for one solution to the problem. Even though we only require a specific kind connection from $L$ to $R$, this implies that a similar kind of connection exists from $R$ to $L$. The following lemma shows that any solution is also a solution to the reverse problem, that is, with the roles of $L$ and $R$ switched.

**Lemma 2.1.** *The* ACTIVATION PROPAGATION *problem is reversible: for any solution, when one vertex of every group in $R$ is activated, there is one group in $L$ in which all vertices are connected to an activated vertex in $R$.*

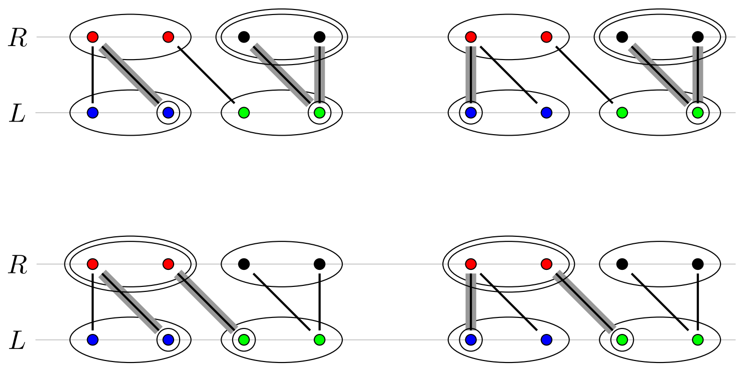

Figure 6: When one vertex is every group of $L$ is activated (indicated by a circle), there is a group in $R$ that is fully activated (doubly circled).

*Proof.* Let $E^*$ be a solution to the ACTIVATION PROPAGATION problem. Suppose we can choose one vertex of every group in $R$ such that there is at least one vertex in every group of $L$ that is not connected to an activated vertex by $E^*$, and denote that set by $R^*$. That means we can choose one vertex of every group in $L$, not connected to any vertex in $R^*$, and let that set be $L^*$. Since $E^*$ is a valid solution to the ACTIVATION PROPAGATION problem, there is one group in $R$ in which all vertices are connected to at least one vertex in $L^*$, contradicting that no vertex in $R^*$ is connected to a vertex in $L^*$. □

The ACTIVATION PROPAGATION problem is therefore symmetric in $L$ and $R$. Figure 7 shows all minimal solutions for a complete bipartite graph where both $L$ and $R$ consist of two groups of two vertices each, up to symmetry. Determining the complexity class of this problem is beyond the scope of this thesis.

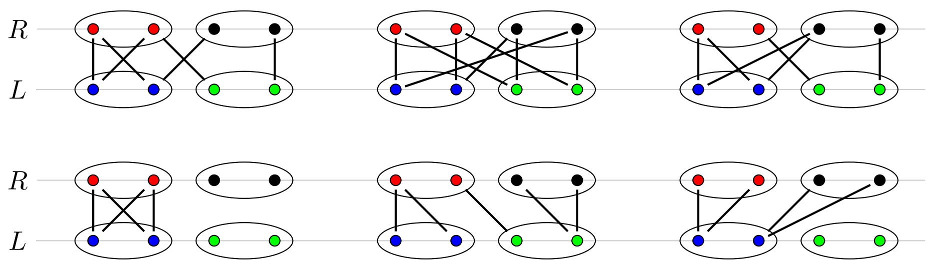

Figure 7: Six solutions to the ACTIVATION PROPAGATION PROBLEM.

## 3 Fixed parameter tractability of MTC

In this section we will show that the minimum temporal connectivity problem is fixed parameter tractable when parameterized by the treewidth and lifetime of the graph. Some simpler variants are already known to be FPT and are therefore only listed without proof.

**Theorem 3.1** ([1])**.** *The $r$-MTC problem is fixed parameter tractable.*

The algorithms for $r$-MTC on a path, tree and on a general graph can be found in A.1, A.2 and A.3 respectively.

**Theorem 3.2** ([1])**.** *The MTC problem is fixed parameter tractable on trees.*

An algorithm for MTC on a tree can be found in A.4.

**Theorem 3.3.** *The MTC problem is fixed parameter tractable for general graphs with treewidth and lifetime as combined parameter.*

For this algorithm we will rely on the separator property of tree decompositions. By recording what temporal paths start and end in the vertices of a node, and which of those paths must be connected together, we can solve MTC using dynamic programming on a nice tree decomposition with arbitrary root. For ease of explanation, we add an extra forget node with $X_1 = \emptyset$ as root node. This node always has index 1.

At every node of the tree decomposition, we keep a list of all possible partial solutions using only the edges introduced in the subtree rooted at that node. Some partial solutions may appear multiple times because we sometimes need to decide beforehand how certain vertices will be connected later on. Of this partial solution, we keep a summary on the active vertices using a graph descriptor, which we will discuss in more detail below. Since some information is lost in the transformation to a graph descriptor, we also keep track of which active vertices still need to be connected to each other within a specified timeframe. In addition, we also keep track of how an undiscovered vertex, when discovering it, would need to be connected to the active vertices in order to be connected to all discovered vertices.

A **graph descriptor** $D$ on $X$ is a set of quadruples $(v_1, t_1, v_2, t_2)$ that describe which vertices in $X$ are connected to each other, including the start and end times of the paths. An example can be found in Figure 8. When discussing node $i$ in a tree decomposition, we usually have $X = X_i$. Note that a graph descriptor is equivalent to a directed multigraph where each edge has a start time and duration.

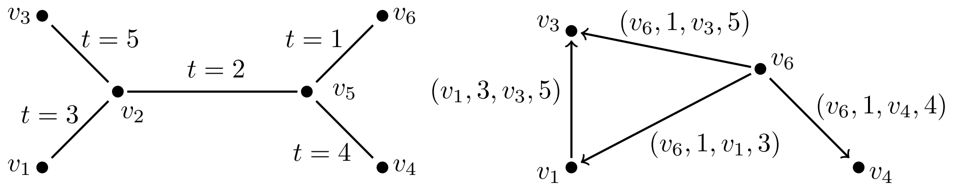

Figure 8: A temporal graph (left) with a corresponding graph descriptor on four of its vertices (right).

We define $T^{in}$ and $T^{out}$ as a set of $k$-tuples containing timestamps, so each can take on $O(2^{\tau^k})$ different values. We define a cost function $f(i, C, D, T^{in}, T^{out})$ that returns the minimum total cost such that the connections in graph descriptor $D$ on $X_i$ are achieved ('done') and all discovered vertices are mutually connected if we add the connections in graph descriptor $C$ on $X_i$ (the 'coming' edges). In addition, for every discovered vertex $v$ and every set of times in $T^{in}$, there is, for at least one value of $j$, a temporal path from $v_j^i$ to $v$ starting no sooner than $t_j^{in}$, and vice versa for $t^{out}$ instead of $t^{in}$. A visualization of this can be found in Figure 9.

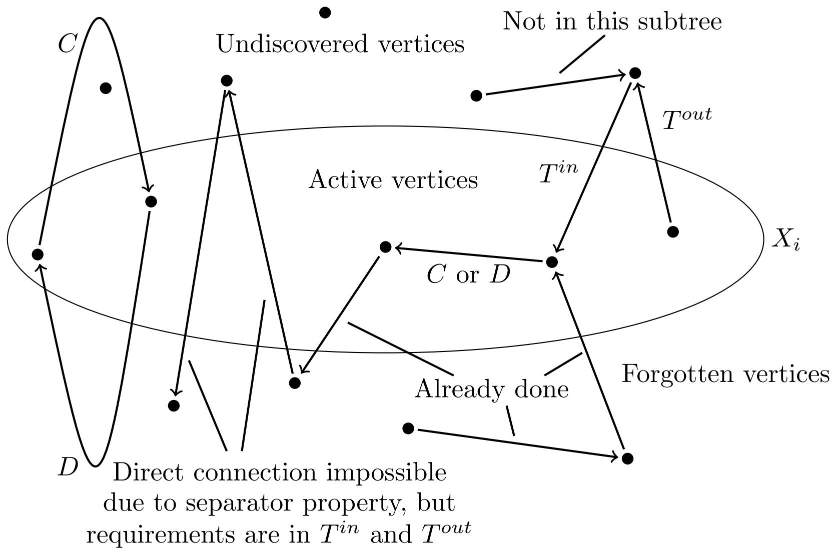

Figure 9: Visualization of the general idea of the algorithm for MTC. Annotations of edges indicate in which part of the cost function guarantees that there is (or will be) a path between the two vertices.

We will only discuss cases where the value of $f$ is finite. Whenever the value of $f$ is undefined, we consider it as $\infty$. For ease of discussion, we will describe the algorithm as if we had a set $A_i$ of quintuples $(C, D, T^{in}, T^{out}, w)$ such that $f(i, C, D, T^{in}, T^{out}) = w$.

## Domination

When two tuples in the same $A_i$ differ in only their value of $w$, we keep the tuple with the lowest $w$ and discard the rest.

This leads us to another issue: many tuples are strictly worse than other tuples, even if they differ in something other than their cost. We will formalize that now through the concept of domination, often used in game theory. When $a$ dominates $b$, we will denote that as $a \succ b$. If $a \succ b$ but not $b \succ a$, $a$ strictly dominates $b$.

In a graph descriptor, an edge $(v_1, t_1, v_2, t_2)$ is dominated if there is a path from $v_1$ to $v_2$ that does not use that edge, starts no sooner than $t_1$ and arrives no later than $t_2$. For two graph descriptors $A$ and $B$, $A \succ B$ if every edge in $B$ is dominated by a path in $A$.

Given two $k$-tuples $t_a^{in}$ and $t_b^{in}$ on the same vertex set, we say that $t_a^{in} \succ t_b^{in}$ if

$t^{in}_{a,i} \geq t^{in}_{b,i}$ for all values of $i$. That is, a $t^{in}k$-tuple can be dominated by another $t^{in}k$-tuple that allows you to enter at the same time or later. Similarly, we have $t^{out}_a \succ t^{out}_b$ if $t^{out}_{a,i} \leq t^{out}_{b,i}$ for all $i$ because $t^{out}_a$ would allow more time to reach the undiscovered vertices.

For the cost value $w$, we have $w_1 \succ w_2$ if $w_1 \leq w_2$.

The set of all tuples which are not strictly dominated by another tuple is a so-called Pareto-optimal set or Pareto-front. All other tuples can be discarded without influencing the final result of the algorithm.

### Leaf node

In a leaf node there is only one discovered vertex of which we have not seen any edges yet. The graph descriptors are therefore empty. Furthermore, every undiscovered vertex must reach $v^i_1$ in $\tau$ time and must be reachable from $v^i_1$ starting at time 1. We thus have $A_i = \{(\emptyset, \emptyset, \{(\tau)\}, \{(1)\}, 0)\}$.

### Introduce vertex node

For ease of notation, we define the $\triangleright$ operator to add an element to a tuple, so $(x_1, \ldots, x_n) \triangleright x$ equals $(x_1, \ldots, x_n, x)$. To make $\triangleright$ work when the left hand side is a set of tuples, we apply $\triangleright x$ to all elements of that set.
Assume without loss of generality that vertex $v^i_k$ is introduced and $k$ is the highest index of all vertices in bag $i$. For every tuple $j$ in $A_{c(i)}$ and every pair of tuples $(t^{in}, t^{out})$ in $T^{in}_j \times T^{out}_j$ , we add $(C_j \cup \{(v^i_k, 1, v_l, t^{in}_l), (v_l, t^{out}_l, v^i_k, \tau) \mid v_l \in X_{c(i)}\}, D_j, T^{in} \triangleright \tau, T^{out} \triangleright 1, w_j)$ to $A_i$.

In the example in Figure 5, at the introduce vertex node for $v_4$, we get

$$A_i = \{(\{(v_3, 1, v_4, 4), (v_4, 1, v_3, 4)\}, \emptyset, \{(4, 4)\}, \{(1, 1)\}, 0)\}\,.$$

### Introduce edge node

Assume wlog that $e = (v^i_1, v^i_2, t)$ is introduced. For every tuple $j$ in $A_{c(i)}$, we add $(C_j, D_j, T^{in}_j, T^{out}_j, w_j)$ and $(C_j, D_j \cup \{(v^i_1, t, v^i_2, t), (v^i_2, t, v^i_1, t)\}, T^{in}_j, T^{out}_j, w_j + w(e))$ to $A_i$. The former corresponds to not using the edge, while the latter includes the edge and updates the done-graph descriptor and total weight values accordingly.

In the example in Figure 5, at the introduce edge node for $(v_3, v_4, 1)$ we get

$$\begin{aligned}A_i = \{&(\{(v_3, 1, v_4, 4), (v_4, 1, v_3, 4)\}, \emptyset, \{(4, 4)\}, \{(1, 1)\}, 0),\\ &(\emptyset, \{(v_3, 1, v_4, 1), (v_4, 1, v_3, 1)\}, \{(4, 4)\}, \{(1, 1)\}, 1),\\ &(\emptyset, \{(v_3, 2, v_4, 2), (v_4, 2, v_3, 2)\}, \{(4, 4)\}, \{(1, 1)\}, 1),\\ &(\emptyset, \{(v_3, 1, v_4, 1), (v_4, 1, v_3, 1), (v_3, 2, v_4, 2), (v_4, 2, v_3, 2)\}, \{(4, 4)\}, \{(1, 1)\}, 2)\}.\end{aligned}$$

Note that the algorithm does not explicitly remove any edges from $C$, but they might be dominated by new edges added to $D$.

**Forget vertex node**

In forget vertex nodes we have to make sure all required connections of the forgotten node are established or will be established later.

Assume wlog that vertex $p = v^i_{|X_{c(i)}|}$ is removed. We do the following for every tuple $j$ in $A_{c(i)}$:

First we will take care of $T^{in}$ and $T^{out}$. For every tuple $t^{in}$ in $T^{in}_j$ and every edge $(v_1, t_1, v_2, t_2)$ in $D_j$ with $v_2 = p$ and $t_2 \leq t^{in}_p$, we add $t^{in}$, in which we replace $t^{in}_{v_1}$ with $\min(t^{in}_{v_1}, t_p)$ and remove $t^{in}_p$, to $T'_{in}$. The construction of $T'_{out}$ is analogous.

We contract the graph $D_j$ at $p$ to form $D'_j$. This means that for any two edges $(v_1, t_1, v_2, t_2)$ and $(v_3, t_3, v_4, t_4)$ with $v_2 = p = v_3$ and $t_2 \leq t_3$, we add $(v_1, t_1, v_4, t_4)$ to $D'_j$ and remove all edges incident to $p$.

Let $C'_j$ be $C_j$ with all edges incident to $p$ removed.

For every edge $e_l = (v_1, t_1, v_2, t_2)$ to (or from) $p$ in $C_j$ that is not dominated by a path in $D_j \cup C'_j$, we create a set $Y_l$ in the following way: for every edge $(v_3, t_3, v_4, t_4)$ in $D$ with $v_4 = v_2 = p$ and $t_2 \geq t_4$ (or $v_3 = v_1 = p$ and $t_1 \leq t_3$), we add $(v_1, t_1, v_3, t_3)$ (or $(v_4, t_4, v_2, t_2)$) to $Y_l$. If this fails for any $e_l$ (such that $Y_l = \emptyset$), that connection can no longer be established and we have to discard $j$. Let $Y$ be the cartesian product of all $Y_l$. Then, for every $(y_1, y_2, \ldots)$ in $Y$, we add

$$(C'_j \cup y_1 \cup y_2 \cup \ldots, D'_j, T'_{in}, T'_{out}, w_j)$$

to $A_i$.

Let us have a look at the forget $v_4$ node in our example (Figure 5). We have already described $A_{c(i)}$ above. We find

$$\begin{aligned} A_i = \{&(\emptyset, \emptyset, \{(1)\}, \{(1)\}, 1), \\ &(\emptyset, \emptyset, \{(2)\}, \{(2)\}, 1), \\ &(\emptyset, \emptyset, \{(2)\}, \{(1)\}, 2)\}. \end{aligned}$$

For the forget $v_2$ node we find

$$\begin{aligned}
A_i = \{&(\{(v_1,3,v_3,4),(v_3,1,v_1,3)\},\emptyset,\{(3,4)\},\{(3,1)\},1),\\
&(\{(v_1,1,v_3,4),(v_3,1,v_1,1)\},\emptyset,\{(1,4)\},\{(1,1)\},1),\\
&(\{(v_1,1,v_3,4),(v_3,1,v_1,3)\},\emptyset,\{(3,4)\},\{(1,1)\},2),\\
&(\{(v_1,1,v_3,4),(v_3,4,v_1,4)\},\emptyset,\{(4,4)\},\{(1,4)\},1),\\
&(\{(v_3,1,v_1,4)\},\{(v_1,3,v_3,4)\},\{(4,4)\},\{(1,4),(3,1)\},2),\\
&(\{(v_3,1,v_1,4)\},\{(v_1,1,v_3,4)\},\{(4,4)\},\{(1,1)\},2),\\
&(\{(v_3,1,v_1,4)\},\{(v_1,3,v_3,4)\},\{(4,4)\},\{(1,1)\},3),\\
&(\{(v_1,1,v_3,3),(v_3,3,v_1,4)\},\emptyset,\{(4,3)\},\{(1,3)\},1),\\
&(\emptyset,\{(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,3),(3,4)\},\{(1,3),(3,1)\},2),\\
&(\{(v_3,1,v_1,4)\},\{(v_1,1,v_3,3)\},\{(4,3),(1,4)\},\{(1,1)\},2),\\
&(\emptyset,\{(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(3,4),(4,3)\},\{(1,1)\},3),\\
&(\{(v_1,1,v_3,4),(v_3,1,v_1,4)\},\emptyset,\{(4,4)\},\{(1,3)\},2),\\
&(\emptyset,\{(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,4)\},\{(1,3),(3,1)\},3),\\
&(\{(v_3,1,v_1,4)\},\{(v_1,1,v_3,3)\},\{(4,4)\},\{(1,1)\},3),\\
&(\emptyset,\{(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,4)\},\{(1,1)\},4)\}.
\end{aligned}$$

**Join node**

For every pair $(l,r)$ in $A_{c_1(i)} \times A_{c_2(i)}$ we create new tuples to add to $A_i$ as follows: For every pair $(t^{in,l}, t^{in,r})$ in $T_l^{in} \times T_r^{in}$, we add $(\min(t_1^{in,l}, t_1^{in,r}), \min(t_2^{in,l}, t_2^{in,r}), \ldots)$ to $T'_{in}$. The construction of $T'_{out}$ is analogous, using max instead of min. We need the most restrictive value here because it holds the requirements to reach forgotten vertices. To make sure the forgotten vertices of both sides can reach each other, we need to add some extra 'coming' edges based on $T^{in}$ and $T^{out}$.

First we will tie $T_l^{out}$ to $T_r^{in}$. We will now consider the ACTIVATION PROPAGATION problem with all tuples in $t^{out,l}$ as groups in $L$ and their elements as vertices in the group. Similarly we take every $t_c^{in,r}$ as a group in $R$ with $t_{c,d}^{in,r}$ as their vertices. The available edges are $(t_{a,b}^{out,l}, t_{c,d}^{in,r})$ for which $t_{a,b}^{out,l} \le t_{c,d}^{in,r}$. For every (minimal) solution $M$ to this problem and every $(t_{a,b}^{out,l}, t_{c,d}^{in,r})$ in $M$, we add $(v_b^i, t_{a,b}^{out,l}, v_d^i, t_{c,d}^{in,r})$ to $A_{l,r}$. The construction of $A_{r,l}$ is analogous.

Finally, for every $(C_{lr}, C_{rl})$ in $A_{lr} \times A_{rl}$, we add $(C_l \cup C_r \cup C_{lr} \cup C_{rl}, D_l \cup D_r, T'_{in}, T'_{out}, w_l + w_r)$ to $A_i$.

There is one join node in our example (Figure 5). We will assume that $v^i = v^{c_1(i)} = v^{c_2(i)} = (v_1, v_3)$. We have already listed $A_{c_1(i)}$ above (the forget $v_2$ node) and we have

$$\begin{aligned}
A_{c_2(i)} = \{&(\{(v_1,1,v_3,1),(v_3,1,v_1,4)\},\emptyset,\{(4,1)\},\{(1,1)\},1),\\
&(\{(v_1,1,v_3,2),(v_3,2,v_1,4)\},\emptyset,\{(4,2)\},\{(1,2)\},1),\\
&(\{(v_1,1,v_3,2),(v_3,1,v_1,4)\},\emptyset,\{(4,2)\},\{(1,1)\},2),\\
&(\{(v_3,1,v_1,1)\},\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(4,1)\},\{(1,1)\},2),\\
&(\emptyset,\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(4,2)\},\{(1,2)\},2),\\
&(\emptyset,\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(4,2)\},\{(1,1)\},3)\}
\end{aligned}$$

which is up to the reader to verify. We find

$$\begin{aligned}
A_i = \{&(\{(v_1,1,v_3,1),(v_3,1,v_1,1)\},\emptyset,\{(1,1)\},\{(1,1)\},2),\\
&(\{(v_1,1,v_3,1),(v_3,1,v_1,1)\},\emptyset,\{(4,1)\},\{(1,1)\},3),\\
&(\{(v_1,1,v_3,1),(v_3,1,v_1,1)\},\{(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,1)\},\{(1,1)\},4),\\
&(\{(v_1,1,v_3,2),(v_3,2,v_1,4)\},\emptyset,\{(4,2)\},\{(1,2)\},3),\\
&(\{(v_1,1,v_3,2)\},\{(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,2)\},\{(1,2)\},4),\\
&(\{(v_1,1,v_3,2),(v_3,1,v_1,1)\},\emptyset,\{(1,2)\},\{(1,1)\},3),\\
&(\{(v_1,1,v_3,2),(v_3,1,v_1,4)\},\emptyset,\{(4,2)\},\{(1,1)\},4),\\
&(\{(v_1,1,v_3,2)\},\{(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,2)\},\{(1,1)\},5),\\
&(\{(v_1,1,v_3,1),(v_3,1,v_1,1)\},\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(1,1)\},\{(1,1)\},3),\\
&(\{(v_1,1,v_3,1),(v_3,1,v_1,1)\},\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(4,1)\},\{(1,1)\},4),\\
&(\{(v_1,1,v_3,1),(v_3,1,v_1,1)\},\{(v_1,2,v_3,2),(v_3,2,v_1,2),(v_1,3,v_3,3),(v_3,3,v_1,3)\},\\
&\qquad\{(4,1)\},\{(1,1)\},5),\\
&(\{(v_3,1,v_1,1)\},\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(1,2)\},\{(1,2)\},3),\\
&(\emptyset,\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(4,2)\},\{(1,2)\},4),\\
&(\emptyset,\{(v_1,2,v_3,2),(v_3,2,v_1,2),(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,2)\},\{(1,2)\},5),\\
&(\{(v_3,1,v_1,1)\},\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(1,2)\},\{(1,1)\},4),\\
&(\emptyset,\{(v_1,2,v_3,2),(v_3,2,v_1,2)\},\{(4,2)\},\{(1,1)\},5),\\
&(\emptyset,\{(v_1,2,v_3,2),(v_3,2,v_1,2),(v_1,3,v_3,3),(v_3,3,v_1,3)\},\{(4,2)\},\{(1,1)\},6),
\end{aligned}$$

Note that many tuples have been simplified or removed because of domination, or removed because they had no solution to the ACTIVATION PROPAGATION problem.

### Solution and correctness

For a given node $i$, vertices are in one of three categories. A vertex is either forgotten, undiscovered, or in the vertex set $X_i$, which we will call the active vertices.

To prove that this algorithm will find an optimal solution, we will define an invariant: At every node $i$, the set $A_i$ corresponds to all possible partial solutions consisting only of edges for which the corresponding introduce edge node is in the subtree rooted at $i$. The cost value is the total weight of all edges in the partial solution. In addition, for every partial solution, it lists all possible (minimal) ways in which it can be extended in order to establish all connections (sometimes in different ways for the same partial solution), as follows:

The connections between forgotten and/or active vertices must be established by the coming edges together with the partial solution, of which the done graph descriptor holds a condensed representation for the active vertices. Connections to be established between forgotten and undiscovered vertices are kept track of by $T^{in}$ and $T^{out}$, and applied when an undiscovered vertex becomes active (introduce vertex node) or forgotten (join node). We do not provide guarantees for connections between two undiscovered vertices or an undiscovered and active vertex, as that will be provided when one vertex becomes forgotten (join or forget vertex nodes) or both become active (introduce vertex node).

Note that this is slightly different from Figure 9, as $T^{in}$ and $T^{out}$ set requirements on connections between active and undiscovered vertices, in order to guarantee connections between forgotten and undiscovered vertices.

For the leaf nodes, introduce vertex nodes and introduce edge nodes it is trivial to check the invariant is kept in place.

At a forget vertex node, one vertex is moved from active to forgotten. Let $p$ be the vertex that is removed. We need to guarantee that connections between $p$ and undiscovered vertices will be established. Therefore, we might need to update up to one value of all $t^{in}$ and $t^{out}$ tuples. Guarantees about connections between $p$ and discovered vertices might have been broken by removing all coming edges incident to $p$. That makes it necessary to fix these broken edges using any path in the partial solution from $p$ to any active vertex (or vice-versa). It suffices to use only a single done edge and add the rest of the required path as coming edge.

Finally, we take a look at the join node. Each forgotten vertex is a forgotten vertex of one child node and an undiscovered vertex for the other child node. Let $p_1$ and $p_2$ be forgotten vertices of join node $i$, of which $p_1$ is forgotten in $c_1(i)$ (and undiscovered in $c_2(i)$) and $p_2$ forgotten in $c_2(i)$. Let $j_1$ be a tuple from $A_{c_1(i)}$ and $j_2$ from $A_{c_2(i)}$. The $T_1^{out}$ value guarantees that for every $t_1^{out}$ in $T_1^{out}$, if we add a path from every vertex $v_j^i$ to $p_2$, starting at $t_{1,j}^{out}$, there is a path from $p_1$ to $p_2$. Similarly, by adding a path from $p_1$ to every $v_j^i$ ending no later than $t_{2,j}^{in}$ we have a connection from $p_1$ to $p_2$, for any $t_{2,j}^{in}$ in $T_2^{in}$. Combining these requirements exactly gives the ACTIVATION PROPAGATION problem, and all solutions for that thus precisely describe all feasible ways to tie $T_1^{out}$ and $T_2^{in}$ together.

By induction on the nodes of the tree decomposition, the invariant therefore holds for all nodes, and particular the root node. As all vertices in the root node are forgotten and the coming graph descriptor is empty, all vertices must be connected by the partial solution. It follows that the minimum total cost to mutually connect all pairs of vertices is $f(1, \emptyset, \emptyset, \emptyset, \emptyset)$, which corresponds to the only non-dominated tuple in $A_1$.

### Running time

**Lemma 3.4.** *The MTC algorithm runs in polynomial time for fixed $\tau$ and $k$.*

*Proof.* A tuple in a $A_i$ set consists of two graph descriptors, two sets of possible times and the cost for that tuple. A graph descriptor is a set of tuples consisting of two vertices and two timestamps, therefore a graph descriptor is most $O(k^2\tau^2)$ in size. Similarly, we can deduce that $T^{in}$ and $T^{out}$ are at most $O(\tau^k)$ in size. Let $R$ be $O(2^{O(k^2\tau^2)})$ and $T$ be $O(2^{\tau^{O(k)}})$. As a result, any $A_i$ can contain at most $O(RT)$ tuples.

We will go over all possible types of nodes. Since Kloks [4] proposed an algorithm to compute a nice tree decomposition with at most $4n$ nodes, we will assume there are at most $O(n)$ nodes of each type except the introduce edge node, of which there are exactly $m$.

The computation for leaf nodes is trivial, yielding $O(n)$ time for all leaf nodes.

There are at most $O(n)$ introduce vertex nodes, each taking up $O(RT^2)$ time to compute each. Therefore, all introduce vertex nodes can be computed in $O(nRT^2)$

time.

For each introduce edge node we only need $O(RT)$ computing time, netting $O(mRT)$ time for all introduce edge nodes together.

A forget vertex node takes up to $O(RT(RT + R + (k\tau)^{k\tau}))$ time, as there can be up to $k\tau$ edges incident to one vertex in a graph descriptor. There is exactly one forget vertex node per vertex, which yields a total computing time of $O(nR^2T^2 + nRT(k\tau)^{k\tau})$ for all forget vertex nodes.

The number of join nodes is at most $O(n)$. The ACTIVATION PROPAGATION problem consists of $O(T)$ vertices and $O(T^2)$ edges and must be solved $O(R^2T^4)$ times. There can be up to $O(2^{T^2})$ solutions to this problem. The total time spent in join nodes is therefore $O(nR^2T^4 2^{T^2})$.

After every step, we need to make sure that dominated edges are removed from the graph descriptors (where 'done' edges can dominated 'coming' edges), $T^{in}$ and $T^{out}$ have their dominated tuples removed, and, after doing so, dominated tuples are removed as well. To achieve the calculated running time, only removing duplicates (also discarding tuples of $A_i$ with higher cost) is required, which takes a time linear in the number of items. For practical purposes, extensive removal of dominated entries is recommended, as even removing a single item from a $T^{in}$ or $T^{out}$ can save a lot of computation time for join nodes.

The total running time is $O(n(R^2T^4 2^{T^2} + RT(k\tau)^{k\tau}) + mRT)$, which is linear in $n$ and $m$ for fixed $\tau$ and $k$. □

*Proof of Theorem 3.3.* Now we have proven that MTC algorithm is correct and runs in polynomial time for graphs with bounded treewidth and lifetime (lemma 3.4), we conclude that the MTC problem is fixed parameter tractable with treewidth and lifetime as combined parameter. □

# 4 Discussion

Even though the practical use cases are limited, the MTC algorithm shows that Minimum Temporal Connectivity belongs in FPT. The algorithm can be easily extended to many variants of the problem by only changing the introduce edge case. These variants include directed graphs, a combination of directed and unidirectional edges, and strict temporal paths.

While we discussed Temporal Matching, it turned out to be out of scope for this thesis. It is likely that a similar algorithm exists for Temporal Matching, which would make Temporal Matching fixed parameter tractable when parameterized by graph lifetime plus treewidth, rather than the cooldown period plus the size of a maximum matching on the underlying graph. We defined Activation Propagation, but did not determine its complexity class nor a solution method other than a trivial exhaustive search. Speeding up the enumeration of all solutions would also positively impact the running time of our MTC algorithm. The question also arises whether finding an optimal edge set for Activation Propagation is polynomially solvable, both for weighted and unweighted graphs.

## References

[1] Kyriakos Axiotis and Dimitris Fotakis. “On the Size and the Approximability of Minimum Temporally Connected Subgraphs”. In: *Proceedings of the 43rd International Colloquium on Automata, Languages and Programming*. 2016, 149:1–149:14.

[2] Marek Cygan, Jesper Nederlof, Marcin Pilipczuk, Michal Pilipczuk, Joham MM van Rooij, and Jakub Onufry Wojtaszczyk. “Solving Connectivity Problems Parameterized by Treewidth in Single Exponential Time”. In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. IEEE. 2011, pp. 150–159.

[3] David Kempe, Jon Kleinberg, and Amit Kumar. “Connectivity and Inference Problems for Temporal Networks”. In: *Journal of Computer and System Sciences* 64(4) (2002), pp. 820–842.

[4] Ton Kloks. *Treewidth: Computations and Approximations*. Springer, 1994.

[5] George B Mertzios, Hendrik Molter, Rolf Niedermeier, Viktor Zamaraev, and Philipp Zschoche. “Computing Maximum Matchings in Temporal Graphs”. In: *37th International Symposium on Theoretical Aspects of Computer Science*. 2020.

[6] Hendrik Molter. *Classic Graph Problems Made Temporal – A Parameterized Complexity Analysis*. Universitätsverlag der TU Berlin, 2020. ISBN: 978-3-7983-3172-3.

[7] Rolf Niedermeier. *Invitation to Fixed-parameter Algorithms*. Vol. 31. OUP Oxford, 2006.

[8] Philipp Zschoche. “A Faster Parameterized Algorithm for Temporal Matching”. In: *Information Processing Letters* 174.106181 (2022).

# A Algorithms

## A.1 $r$-MTC on a path

The $r$-MTC problem can be solved using dynamic programming when the graph is a path. Assume that $r$ is one of the two endpoints. Starting at $r$, we traverse the graph, keeping track of the lowest cost to reach a vertex within $t$ timesteps for $1 \leq t \leq \tau$. More formally, with vertices numbered from $r = 1$ to $n = |V|$, we define

$$f(v,t) = \begin{cases} 0 & \text{if } v = r = 1 \\ \infty & \text{if } t \leq 0 \\ \min \begin{cases} f(v, t-1) \\ f(v-1, t) + w((v-1, v, t)) \end{cases} & \text{otherwise} \end{cases} \tag{1}$$

where $w(e) = \infty$ if $e$ does not exist. The value of the solution is $f(|V|, \tau)$, where the edges of the solution can be found by backtracking. If $r$ is not an endpoint of the path, we run this algorithm twice by cutting the graph through $r$, making $r$ an endpoint on both halves of the graph.

## A.2 $r$-MTC on trees

This is an adaption of the algorithm described in section A.1. For this algorithm to work on trees, we first need to reverse the working direction to work from the leaves up to $r$, which we will use as root. We define $S(v)$ as the set of all direct children of node $v$ in a tree. For any leaf node $v$, $S(v) = \emptyset$. We define two separate functions for calculating the cost of connecting one child node and all child nodes. For $w \in S(v)$ we define

$$f(v,w,t) = \begin{cases} \infty & \text{if } t > \tau \\ \min \begin{cases} f(v, w, t+1) \\ f(w, t) + w((v, w, t)) \end{cases} & \text{otherwise} \end{cases} \tag{2}$$

where

$$f(v,t) = \sum_{w \in S(v)} f(v, w, t). \tag{3}$$

For leaf nodes, this sum is empty and thus sums to zero, as desired. The value of the optimal solution is $f(r, 0)$ and can be found in $O(n\tau)$ time.

## A.3 $r$-MTC using treewidth

This is an adaption of the algorithm discussed in section 4.3 of [1] using $O(nk^2 3^k (L + k)^{k+1})$ time. We denote the set of all forgotten vertices of bag $i$ as $F_i$. Given a (temporal) graph $G$, we take a *nice tree decomposition* $H$ of treewidth $k$. Let $X_i$ be a bag of $H$, $a_j \in \{0, 1\}$, and $1 \leq t \leq \tau$. We define $f(i, a_1, t_1, \ldots, a_k, t_k)$ as the minimum cost it takes to connect vertices in such a way that the following holds: For all vertices $w$ for which $w \in F_i$ or $w = v^i_{j'}$ and $a_{j'} = 0$, there is, for some $j$, a temporal path from $v^i_j$ to $w$ starting no sooner than $t_j$. The path must also end no later than $t_{j'}$ in the latter case. Some parameters of $f$ may be undefined. The total minimal cost of the optimal solution to $r$-MTC is $f(1, 1, 1)$, as we start at the root at time 1.

A vertex $v_j^i$ in a specific node can be in 2 states: It is not connected ($a_j = 1$) and requires an entry before a specific timestep $t_j$, or it is connected ($a_j = 0$) through another vertex, no later than $t_j$. As a result, we can require that $f(i, \dots, 1, t_j, \dots) \geq f(i, \dots, 1, t_j{+}1, \dots)$ and $f(i, \dots, 0, t_j, \dots) \leq f(i, \dots, 0, t_j{+}1, \dots)$ We split cases based on what bag $i$ is:

**Leaf node** Vertices in leaf nodes are not yet connected to anything else, so $f(i, 0, t_1)$ $= \infty$ and $f(i, 1, t_1) = 0$.

**Introduce node** Assume wlog that $v_p^i$ was introduced. Since the new vertex is not yet connected to any other vertex, we can choose $f(i, \dots, 0, t_p, \dots) = \infty$ and $f(i, \dots, a_{p-1}, t_{p-1}, 1, t_p, a_{p+1}, t_{p+1}, \dots) = f(c(i), \dots, a_{p-1}, t_{p-1}, a_{p+1}, t_{p+1}, \dots)$.

**Forget node** Assume wlog that $v_p^i$ was removed. The vertex removed must be connected to another vertex. We thus have $f(i, \dots, a_{p-1}, t_{p-1}, a_{p+1}, t_{p+1}, \dots) = f(c(i), \dots, a_{p-1}, t_{p-1}, 0, \tau, a_{p+1}, t_{p+1}, \dots)$.

**Join node** Any vertex that is connected in a join node, must be connected in exactly one of its child nodes. We therefore get $f(i, a_1, t_1, \dots) = \min\{f(c_1(i), a_{1,1}, t_1, \dots) + f(c_2(i), a_{1,2}, t_1, \dots) \mid a_{j,1} + a_{j,2} = a_j + 1\}$.

**Introduce edge node** Let $e = ((u, w), t_e)$ be the edge associated with this node. Since both endpoints of the edge must be in the associated introduce edge node, we can assume wlog that $u = v_1^i$ and $w = v_2^i$. The edge can be used in either direction or not at all, and the originating vertex can be either connected or not. We therefore split based on four cases:

$$f(i, 1, t_1, 1, t_2, \dots) = f(c(i), 1, t_1, 1, t_2, \dots) \tag{4}$$

$$f(i, 1, t_1, 0, t_2, \dots) = \min \begin{cases} f(c(i), 1, t_1, 0, t_2, \dots) \\ f(c(i), 1, t_1, 1, t_3, \dots) + c(e) \text{ if } t_1 \leq t_e \leq t_2, t_3 \end{cases} \tag{5}$$

$$f(i, 0, t_1, 1, t_2, \dots) = \min \begin{cases} f(c(i), 0, t_1, 1, t_2, \dots) \\ f(c(i), 1, t_3, 1, t_2, \dots) + c(e) \text{ if } t_2 \leq t_e \leq t_1, t_3 \end{cases} \tag{6}$$

$$f(i, 0, t_1, 0, t_2, \dots) = \min \begin{cases} f(c(i), 0, t_1, 0, t_2, \dots) \\ f(c(i), 1, t_3, 0, t_2, \dots) + c(e) \text{ if } t_2 \leq t_e \leq t_1, t_3 \\ f(c(i), 0, t_1, 1, t_3, \dots) + c(e) \text{ if } t_1 \leq t_e \leq t_2, t_3. \end{cases} \tag{7}$$

## A.4 MTC on a tree

We can adapt the algorithms from the previous sections to work for MTC as well. The key observation is that we need a path from an arbitrary root $r$ to all leaves and a path from all leaves to $r$, with some extra constraints on the timing to make sure all leaves are also temporally connected to each other. The value of $f(v, t, t')$ represents the minimum total cost of all edges below $v$ such that we can reach all vertices below $v$ starting at time $t$, all descendants of $v$ can reach $v$ within the first $t'$ timesteps and all descendants of $v$ are all mutually temporally connected. First, we set $f(\ldots) = \infty$ whenever one of the time parameters is out of bounds. Then, for $v \in V, w \in S(v), 1 \leq t, t' \leq \tau$, we define

$$f(v,w,t,t') = \min \begin{cases} f(v,w,t+1,t') \\ f(v,w,t,t'-1) \\ f(w,t,t') + w((v,w,t)) & \text{if } t = t' \\ f(w,t,t') + w((v,w,t)) + w((v,w,t')) & \text{if } t \neq t'. \end{cases} \tag{8}$$

Combining the cost functions for each edge is a bit more complicated than simply summing over all children with the same time parameters. This is because the descendants in different branches need to be able to reach each other too. This might fail when $t < t'$, but only when that happens for more than one child node.

$$f(v,t,t') = \begin{cases} \sum_{w \in S(v)} f(v,w,t,t') & \text{if } t \geq t' \\ \min_{w \in S(v)} f(v,w,t,t') + \sum_{w' \in S(v), w \neq w'} f(v,w',t',t) & \text{if } t < t'. \end{cases} \tag{9}$$

The value of the solution is $\min_{1 \leq t,t' \leq \tau} f(r,t,t')$ and can be found in $O(n\tau^2)$ time.