Information and Computation 293 (2023) 105063

Contents lists available at ScienceDirect

Information and Computation

journal homepage: www.elsevier.com/locate/yinco

A faster algorithm for converting simple temporal networks with uncertainty into dispatchable form

Luke Hunsberger [a], Roberto Posenato [b,*]

[a] Department of Computer Science, Vassar College, Vassar, NY, USA
[b] Department of Computer Science, University of Verona, Verona, Italy

ARTICLE INFO

Article history:
Received 23 May 2022
Received in revised form 25 May 2023
Accepted 8 June 2023
Available online 14 June 2023

Keywords:
Temporal constraint networks
Contingent durations
Dispatchable network
Scheduling algorithm

ABSTRACT

A Simple Temporal Network with Uncertainty (STNU) is a data structure for reasoning about time constraints on actions that may have uncertain durations. An STNU is *dispatchable* if it can be executed in real-time with minimal computation 1) satisfying all constraints no matter how the uncertain durations play out and 2) retaining maximum flexibility. The fastest known algorithm for converting STNUs into dispatchable form runs in $O(n^3)$ time, where n is the number of timepoints. This paper presents a faster algorithm that runs in $O(mn + kn^2 + n^2 \log n)$ time, where m is the number of edges and k is the number of uncertain durations. This performance is particularly meaningful in fields like Business Process Management, where sparse STNUs can represent temporal processes or plans. For sparse STNUs, our algorithm generates dispatchable forms in time $O(n^2 \log n)$, a significant improvement over the $O(n^3)$-time previous fastest algorithm.

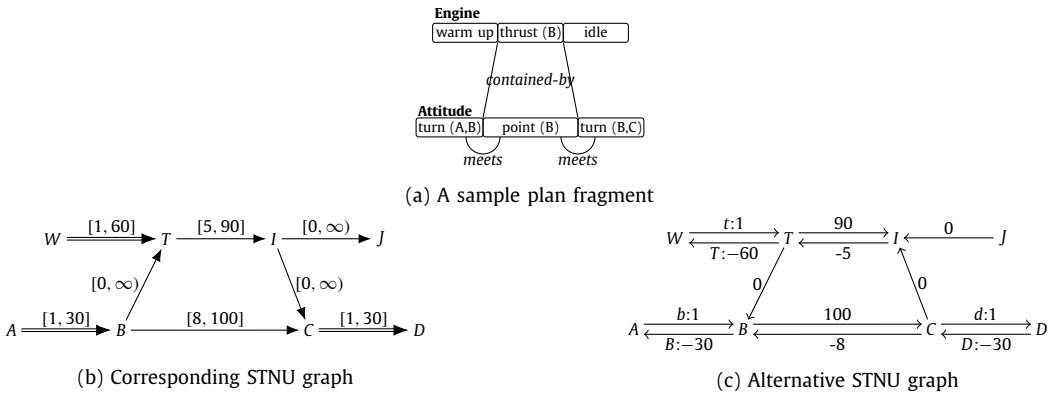
1. Introduction

In many sectors of real-world industry, it is necessary to plan and schedule tasks allocated to multiple agents participating in complex processes. Temporal planning aims to determine the order of task execution while respecting temporal constraints such as, for example, release times, maximum durations, and deadlines [13]. To build feasible plans, a temporal planner must be able to do quantitative temporal reasoning. The literature includes many formalisms of temporal reasoning that differ in their expressiveness and computational complexity [13,2].

Over the past twenty-five years, several major application developers have highlighted the need for (1) the explicit representation of actions with uncertain durations; (2) efficient algorithms for determining whether plans involving such actions are controllable; and (3) efficient algorithms for converting such plans into a form that enables them to be executed in real-time with minimal computation, while preserving maximum flexibility. For example, researchers working on NASA's *Deep Space One* and *Remote Agent* projects stressed that, during the real-time execution of a plan, *"The latency issue affects the form of updates that are propagated through the network, and reinforces the need for an efficient executive component. This motivates the use of limited propagation, which in turn introduces the need to augment the network with implied links ... which are also useful for supporting stronger forms of propagation that are required to handle scheduling with uncontrollable events [e.g., actions with uncertain durations]"* [28]. More recently, the need to address these kinds of issues has been raised in a variety of domains involving

* Corresponding author.
E-mail addresses: hunsberger@vassar.edu (L. Hunsberger), roberto.posenato@univr.it (R. Posenato).

https://doi.org/10.1016/j.ic.2023.105063
0890-5401/



(a) A sample plan fragment

(b) Corresponding STNU graph

(c) Alternative STNU graph

Fig. 1. A plan fragment for an autonomous spacecraft [28] and two equivalent representations of its temporal aspects as an STNU.

autonomous robots [16,6], rovers [4,35,1], and underwater vehicles [37,19,3,1,12], as well as more broadly in the field of Business Process Management [7,20,22,21,10,14,15,31,11].

Over the past two decades, numerous researchers have made significant theoretical and practical progress in all of these areas, as follows: (1) providing the theoretical foundations for Simple Temporal Networks with Uncertainty (STNUs) which can be used to explicitly represent actions with uncertain durations [26,17]; (2) providing polynomial-time algorithms for determining whether any given STNU is *dynamically controllable* (DC) [27,38,23,24,30,5]; and (3) providing polynomial-time algorithms for converting an STNU into so-called *dispatchable* form [29,39,36,24,25].

For a simple illustration, Fig. 1a shows a sample plan fragment for an autonomous spacecraft that consists of two timelines: one for the engine, and one for its attitude, where the timing of the *thrust*, *point* and *idle* actions are assumed to be under the full control of the execution module, whereas the *warm_up* and *turn* actions are assumed to have uncontrollable, but bounded durations. In other words, the execution module controls when instances of these actions start, but only observes their completion in real time, as they occur. The temporal aspects of this plan can be represented by an STNU, shown in Fig. 1b, where timepoints such as A, B, and C represent the starting or ending times of actions, arrows labeled by intervals represent temporal constraints to be satisfied, and double arrows labeled by intervals represent so-called *contingent links*. For each contingent duration, the execution module is presumed to control the starting time of the action, but not its duration. However, it may react to observations of contingent executions in real-time. Fig. 1c shows an alternative, equivalent representation of that STNU.

For any STNU, it is important to determine whether it is *dynamically controllable* (i.e., whether it is possible, in real-time, to schedule all of its non-contingent timepoints such that all constraints will necessarily be satisfied no matter what durations turn out for the contingent links). Several polynomial-time algorithms are available to solve this so-called DC-checking problem [27,23,24,5]. However, once an STNU is known to admit a dynamic scheduler, it is also important to be able to compute one efficiently. For maximum flexibility and minimal space and time requirements, a dynamic scheduler for an STNU is typically computed *incrementally,* in real-time, so that it can react to observations of contingent executions as they occur. An efficient dynamic scheduler can be realized by first transforming an STNU into a *dispatchable* form [29,39]. Then, to execute the dispatchable STNU, it suffices to maintain *time-windows* for each timepoint and, as each timepoint X is executed, only updating time-windows for neighbors of X in the graph. Dispatchable STNUs are very important in applications that demand quick responses to observations of contingent events. Furthermore, applications that involve frequent replanning require efficient algorithms for converting STNUs into dispatchable forms.

Previously, in the literature, the fastest algorithm for computing the dispatchable form of a dynamically controllable STNU is Morris' 2014 algorithm, which runs in $O(n^3)$ time, where n is the number of timepoints [24]. This paper presents a faster, $O(mn + kn^2 + n^2 \log n)$-time algorithm for converting dynamically controllable STNUs into dispatchable form, where m is the number of temporal constraints, and k is the number of contingent links. Such an improvement in time complexity is significant for practical applications (e.g., modeling business processes) where the networks are typically sparse. For example, if $m = O(n \log n)$ and $k = O(\log n)$, then our algorithm runs in $O(n^2 \log n) \ll O(n^3)$ time. Similarly, if $m = O(n^{1.5})$ and $k = O(\sqrt{n})$, then our algorithm runs in $O(n^{2.5}) \ll O(n^3)$ time. Furthermore, our algorithm achieves an order-of-magnitude speed-up in practice because it typically inserts an order of magnitude fewer new edges into the network than does Morris' algorithm. The paper includes a proof of correctness for our algorithm and an empirical evaluation that demonstrates its better performance over existing public benchmarks.

2. Background

A *Simple Temporal Network* (STN) is a data structure for representing and reasoning about time constraints on actions, where each action is represented as a pair of instantaneous events called *timepoints*, representing the starting and ending times of the action. Formally, an STN is a pair $(\mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ is a set of real-valued variables (timepoints), and $\mathcal{C}$ is a set of binary difference constraints of the form, $Y - X \leq \delta$, where $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$ [9]. The graph of an STN is a pair

Table 1
A high-level comparison of the two fastest DC-checking algorithms for STNUs.

Algorithm	Time-complexity	Problem edges	Rules	Prop dirn	Prop along	Pot func?
Morris 2014	$O(n^3)$	Neg. edges	Table 2	Bkwd	Non-neg. edges	No
RUL$^-$ 2018	$O(mn + k^2n + kn\log n)$	UC edges	Table 3	Bkwd	LO-edges	Yes

$(\mathcal{T}, \mathcal{E}_o)$, where for each $(Y - X \leq \delta)$ in $\mathcal{C}$, there is a labeled directed edge $X \xrightarrow{\delta} Y$ in $\mathcal{E}_o$. In this paper, the constraints and edges associated with an STN are called *ordinary* constraints and edges.[1] For convenience, ordinary edges may be notated as (X, δ, Y). An STN is *consistent* (i.e., has a solution) if and only if its graph has no negative cycles [9]. Finally, two STNs over the same set of timepoints are called *equivalent* if they admit the same set of solutions.

A Simple Temporal Network with Uncertainty augments an STN to accommodate actions with uncertain durations [26]. An STNU is a triple $(\mathcal{T}, \mathcal{C}, \mathcal{L})$, where $(\mathcal{T}, \mathcal{C})$ is an STN, and $\mathcal{L}$ is a set of *contingent links,* each having the form (A, x, y, C), where $0 < x < y < \infty$, and $A, C \in \mathcal{T}$. Each (A, x, y, C) represents an uncertain duration where $x \leq C - A \leq y$. A is called the *activation* timepoint; C, the *contingent* timepoint; and $\Delta_C = y - x$, the uncertainty in the link's duration. Although distinct contingent links may share an activation timepoint, they must have distinct contingent timepoints. We let $\mathcal{T}_C$ denote the set of contingent timepoints, and $\mathcal{T}_X = \mathcal{T} \backslash \mathcal{T}_C$ the set of *executable* timepoints.

The graph for an STNU $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ is a pair, $(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, where $(\mathcal{T}, \mathcal{E}_o)$ is the graph for the STN $(\mathcal{T}, \mathcal{C})$, and the contingent links in $\mathcal{L}$ correspond to edges in $\mathcal{E}_\ell \cup \mathcal{E}_u$. Specifically, for each $(A, x, y, C) \in \mathcal{L}$, there are two edges: a *lower-case* (LC) edge, and a *upper-case* (UC) edge. The *lower-case* (LC) edge $A \xrightarrow{C:x} C$ in $\mathcal{E}_\ell$ represents the *uncontrollable possibility* that the duration $C - A$ may be as low as x. The *upper-case* (UC) edge $C \xrightarrow{C:-y} A$ in $\mathcal{E}_u$ represents the *uncontrollable possibility* that the duration $C - A$ may be as high as y.

Such edges may be notated as $(A, c{:}x, C)$ and $(C, C{:}-y, A)$, respectively.[2] Fig. 1c shows the STNU graph corresponding to the plan fragment from Fig. 1a. It contains three contingent links, where W, A and C are the activation timepoints, and T, B, and D are the contingent timepoints.

Following the literature, we let $n = |\mathcal{T}|$, $m = |\mathcal{C}|$, and $k = |\mathcal{L}|$; $\mathcal{E}_{\ell o} = \mathcal{E}_\ell \cup \mathcal{E}_o$ the set of *LO-edges*; $(\mathcal{T}, \mathcal{E}_{\ell o})$, the *LO-graph*; $\mathcal{E}_{ou} = \mathcal{E}_o \cup \mathcal{E}_u$, the set of *OU-edges*; and $(\mathcal{T}, \mathcal{E}_{ou})$, the *OU-graph.*

If $\mathcal{L}$ is empty (i.e., no contingent links), then the STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \emptyset)$ reduces to the STN $(\mathcal{T}, \mathcal{C})$. In addition, for any STNU, the corresponding LO-graph and OU-graph may be viewed as STNs by ignoring the alphabetic labels on their edges.

2.1. Dynamic controllability

Although the durations of contingent links are typically not known in advance, a *dynamic strategy* can be used to govern the execution of the *executable* timepoints (i.e., those in $\mathcal{T}_X$). Unlike fixed solutions for STNs, a dynamic strategy for an STNU may react in real-time to observe executions of contingent timepoints (i.e., those in $\mathcal{T}_C$), but its decisions about when to execute timepoints in $\mathcal{T}_X$ cannot depend on advance knowledge of future events. If there exists a dynamic strategy for executing the timepoints in $\mathcal{T}_X$ that guarantees that all ordinary constraints will be satisfied no matter how the durations of the as-yet-unexecuted contingent links turn out, then the STNU is said to be *dynamically controllable* (DC) [26,17].

Given its practical importance, researchers have recently presented several efficient DC-checking algorithms, all of which use (sound) rules for generating new edges that effectively *bypass* certain kinds of *problematic* edges in the STNU graph. For each algorithm, if the generation of bypass edges does not yield a certain kind of negative cycle, then the STNU is necessarily DC. The algorithms differ in the kinds of edges they view as problematic, the edge-generation rules they use, the direction of propagation (forward or backward), whether they need to compute and update a *potential function,* and, if so, the edges from which the potential function is derived. The two fastest DC-checking algorithms, the $O(n^3)$-time algorithm due to Morris [24] and the $O(mn + k^2n + kn\log n)$-time algorithm due to Cairo et al. [5], will be most relevant for this paper. Their high-level features are compared in Table 1.

2.1.1. The Morris 2014 DC-checking algorithm

Morris [24] introduced an $O(n^3)$-time DC-checking algorithm that uses the edge-generation rules shown in Table 2. (The edges generated by the rules are shown as dashed.) The algorithm views *negative* edges (i.e., edges having a negative weight) as problematic. It propagates *backward* from so-called *negative nodes* (i.e., nodes having one or more incoming negative edges) along paths in the *LO-graph,* aiming to generate non-negative edges to effectively bypass the negative edges.

Fig. 2 shows sample propagations by the Morris 2014 algorithm. Each starts from the source node of a *negative* edge (shown as red and thick) and proceeds *backward* along *non-negative* LO-edges. Since all edges (other than the first one)

[1] In the literature, STNs and their graphs are sometimes defined using an interval notation, where a constraint $Y - X \in [a, b]$, sometimes called a *requirement link,* is equivalent to the pair of ordinary constraints, $Y - X \leq b$ and $X - Y \leq -a$; and the link $X \xrightarrow{[a,b]} Y$ is equivalent to the pair of ordinary edges, $X \underset{-a}{\overset{b}{\rightleftarrows}} Y$. The interval notation is not used in this paper because it frequently requires the use of ∞ and $-\infty$ to represent "no constraint". In addition, each interval-based constraint has two representations (e.g., $Y - X \in [a, b]$ is equivalent to $X - Y \in [-b, -a]$).

[2] In the literature, and as illustrated in Fig. 1b, contingent links in STNU graphs are sometimes represented using intervals. This paper prefers the labeled LC and UC edges because they enable a more intuitive form for constraint-propagation rules.

Table 2
Edge-generation rules used by Morris 2014 [24].

Rule	Graphical representation	Conditions
(NC)	$X \xrightarrow{v} Y \xrightarrow{w} W$; $X \dashrightarrow W$: $v+w$	(none)
(UC)	$X \xrightarrow{v} Y \xrightarrow{C:w} A$; $X \dashrightarrow A$: $C:v+w$	(none)
(LC)	$A \xrightarrow{c:x} C \xrightarrow{w} X$; $A \dashrightarrow X$: $x+w$	$w < 0$
(CC)	$A \xrightarrow{c:x} C \xrightarrow{K:w} B$; $A \dashrightarrow B$: $K:x+w$	$K \neq C$, $w < 0$
(LR)	$X \xrightarrow{C:w} A \xrightarrow{c:x} C$; $X \dashrightarrow A$: w	$w \geq -x$

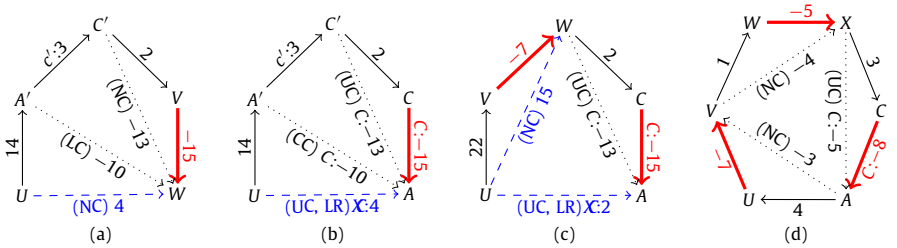


Fig. 2. Sample Propagations by the Morris 2014 Algorithm. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

are non-negative, Dijkstra's algorithm [8] can be used to guide the traversal of shortest paths without needing a potential function.

In Fig. 2a, the NC (*No Case*) and LC (*Lower Case*) rules (shown in parentheses) are used to generate the non-negative *bypass* edge $(U, 4, W)$, shown as blue and dashed. The intermediate edges, shown as dotted, are computed along the way, but not inserted into the graph. More generally, a *single* run of Dijkstra can be used to generate bypass edges for multiple *ordinary* negative edges incoming to a single node.

Fig. 2b is similar to 2a, except that the initial edge $(C, C{:}-15, A)$ and the intermediate (dotted) edges are UC edges. The intermediate UC edges, generated by the UC (*Upper Case*) and CC (*Cross Case*) rules, correspond to conditional constraints, called *waits*. For example, the edge $(A', C{:}-10, A)$ corresponds to a *wait* that can be glossed as: *"While C remains unexecuted, A' must wait at least* 10 *after A."* Finally, the UC and LR (*Label Removal*) rules combine to generate the bypass edge $(U, 4, A)$, shown as blue and dashed.

This approach to generating bypass edges does not generalize to cases in which the negative edges coming into a node include both ordinary and UC edges or multiple UC edges. Therefore, the algorithm first does an $O(n)$-time pre-processing step to ensure that the negative edges coming into a negative node are *either* all ordinary edges *or* a single UC edge, but not both.

Fig. 2c shows that back-propagation from one negative edge can be interrupted by an encounter with another negative edge. Here, back-propagation from C is interrupted by the negative edge $(V, -7, W)$. Once edge $(V, -7, W)$ is bypassed by the (blue, dashed) edge $(U, 15, W)$, back-propagation from C can continue, eventually generating the (blue, dashed) edge, $(U, 2, A)$. More generally, any time the processing of a negative node X encounters another negative node Y, the processing of X is interrupted until the processing of Y is completed. A cycle of such interruptions, which is easy to track, necessarily corresponds to a cycle of (negative) intermediate (dotted) edges, as shown in Fig. 2d, which signals that the STNU is not DC. If all negative nodes can be fully processed without encountering such a cycle, then the network is guaranteed to be DC.

Since there are at most $O(n)$ negative nodes, and the processing of each negative node can be done in $O(\hat{m} + n \log n)$ time, where $\hat{m} = O(n^2)$ is the number of edges in the graph at the end of the algorithm, the overall complexity is $O(n^3)$.

2.1.2. The RUL$^-$ *DC-checking algorithm*

Cairo et al. [5] introduced a DC-checking algorithm, called RUL$^-$, that views *UC edges* as problematic. From each UC edge E, it propagates *backward* along *LO-edges,* using the rules from Table 3, aiming to generate *ordinary* edges that bypass E.

Because LO-edges can be negative, the algorithm computes and updates a solution for the LO-graph, viewed as an STN. As in Johnson's algorithm [8], it uses this solution as a *potential function* to enable Dijkstra-like traversals of shortest paths. The potential function is initialized by an $O(mn)$-time call to Bellman-Ford [8], and periodically updated by an incremental algorithm similar to the one of Ramalingam et al. [34]. Since there are k UC edges, and it adds at most kn edges, the overall complexity reduces to $O(mn + k^2 n + kn \log n)$, a significant improvement over $O(n^3)$, especially for sparse graphs.

Table 3
Edge-generation rules used by RUL$^-$ [5].

Rule	Graphical representation	Applicability conditions
(R)	$P \xrightarrow{v} Q \xrightarrow{w} C$; $v + w$	$Q \in \mathcal{T}_X$, $w < \Delta_C$, $C \in \mathcal{T}_C$
(L)	$A_j \xrightarrow{c':x'} C' \xrightarrow{w} C$; $x' + w$	$C' \not\equiv C$, $w < \Delta_C$, $C \in \mathcal{T}_C$
(U_{lp})	$P \xrightarrow{v} C \xrightarrow{C:-y} A$; $v - y$	$(A, x, y, C) \in \mathcal{L}$, $v \geq \Delta_C$
(U_{nlp})	$P \xrightarrow{v} C \xrightarrow{C:-y} A$; $-x$	$(A, x, y, C) \in \mathcal{L}$, $v < \Delta_C$

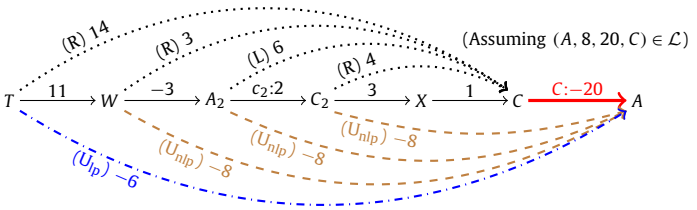


Fig. 3. RUL$^-$ generating bypass edges for the (thick, red) UC edge.

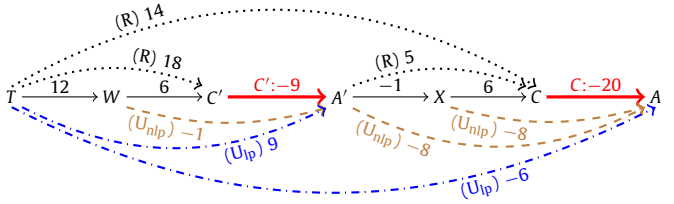


Fig. 4. RUL$^-$ dealing with an interrupting UC edge, assuming that the relevant contingent links are $(A', 1, 9, C')$ and $(A, 8, 20, C)$.

Fig. 3 illustrates the two-phase processing of a UC edge $E = (C, C{:}-20, A)$ by the RUL$^-$ algorithm. The first phase propagates backward from C along LO-edges, using the R (*Relax*) and L (*Lower*) rules from Table 3 to generate ordinary edges terminating at C (shown as dotted edges). The applicability conditions of the R and L rules ensure that this phase stops when an edge with length at least $\Delta_C = 20 - 8 = 12$ has been generated: here, with the edge $(T, 14, C)$. Then, for each newly generated (dotted) edge $\hat{E}$, the second phase combines $\hat{E}$ with E using either the U_{nlp} or U_{lp} rule to generate bypass edges (shown as dashed or dash-dotted). (U is for *Upper*; *nlp* is for *non-length-preserving*; and *lp* is for *length-preserving*.) The applicability conditions for these rules ensure that every bypass edge will have length $-x = -8$ (shown in brown, dashed), except the last one (shown in blue, dash-dotted), which has length $14 - 20 = -6$.

In general, the back-propagation in the first phase is guided by a Dijkstra-like traversal of LO-paths, which is made possible by using the above-mentioned potential function to reweight the edges in the LO-graph. If, as is typical, the bypass edges inserted into the graph during the second phase introduce new shortest paths into the LO-graph, then the potential function for the LO-graph must be incrementally updated, as previously mentioned, in preparation for processing the next UC edge.

The RUL$^-$ algorithm's processing of one UC edge can be interrupted by another, as shown in Fig. 4, where the first-phase back-propagation from C is interrupted by the UC edge $(C', C'{:}-9, A')$. Processing this UC edge generates the (blue, dashed) bypass edge $(T, 9, A')$, after which the first-phase processing of $(C, C{:}-20, A)$ continues, generating the (dotted) edge $(T, 14, C)$ and, in the second phase, the bypass edge $(T, -6, A)$. Similar to the Morris 2014 algorithm, a cycle of such interruptions would imply that the STNU was not DC, but if all UC edges can be fully processed without encountering such a cycle, then the network must be DC.

2.2. Dispatchability

Although the DC property for STNUs is important, knowing that a dynamic strategy exists is not the same as being able to efficiently compute one. For one thing, a dynamic strategy for an STNU may have exponentially many branches, and so it may not be practical to compute it in advance. Instead, such strategies are typically computed *incrementally,* in real-time, which enables execution decisions to react to observations of contingent executions as they occur. Such considerations led researchers to define the *dispatchability* of temporal networks, and to develop algorithms for converting STNUs into a dispatchable form. Dispatchable STNUs are very important in applications that demand quick responses to observations of

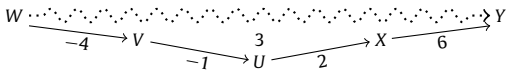


Fig. 5. A vee-path of length 3 in an STN.

contingent events. The rest of this section summarizes relevant results from the literature on dispatchability—for STNs and STNUs.

2.2.1. STN dispatchability

Although solutions for consistent STNs can be efficiently computed in advance, it is often desirable to compute solutions incrementally, in real-time, to preserve as much flexibility as possible during execution. Toward that end, Tsamardinos et al. [39] defined a *real-time dispatching* (RTD) algorithm that incrementally *attempts* to compute a solution for an STN in real-time, using only *local* propagation. For each timepoint, the RTD algorithm maintains a *time-window* that contains possible values for it. The algorithm iteratively selects a timepoint X to be executed next and a time v at which to execute it. It then updates the time-windows for *neighbors* of X (i.e., timepoints that are connected to X by a *single* edge—hence, *local*). For example, if X is executed at time 10, then an outgoing non-negative edge $(X, 5, Y)$ generates an upper bound of $10+5=15$ for Y, and an incoming negative edge $(W, -3, X)$ generates a lower bound of $10+3=13$ for W. In addition, once X has been executed at time v, all remaining unexecuted timepoints must be executed at or after v. (No going backward in time.) It follows that a timepoint is *enabled* for execution only if all of its outgoing negative edges terminate at already-executed timepoints. Because it performs only one update per edge, the RTD algorithm runs in $O(m)$ time, making it very practical.

Tsamardinos et al. [39] then defined an STN graph $\mathcal{G}$ to be *dispatchable* if *every* run of the RTD algorithm on $\mathcal{G}$ is guaranteed to generate a solution for $\mathcal{G}$—no matter how the choices of X and v are made, as long as the time-window bounds and enablement requirements are observed. Muscettola et al. [29] then gave an $O(mn + n^2 \log n)$-time algorithm, called *fastDispatchMin* (FDM), that converts any consistent STN into *minimal* dispatchable form (i.e., an equivalent dispatchable STN having the fewest edges). More recently, Morris [25] provided a *graphical* characterization of STN dispatchability in terms of *vee-paths*. A vee-path is any path that consists of zero or more negative edges, followed by zero or more non-negative edges. An STN is *vee-path-complete* (VPC) if, whenever there is a path from any X to any Y, there is a shortest path from X to Y that is a vee-path.

Theorem 1 *(Morris [25]). A consistent STN is dispatchable if and only if it is vee-path-complete.*

The intuition behind this result is illustrated in Fig. 5. Given that the RTD algorithm only executes a timepoint if it has no negative outgoing edges to *unexecuted* timepoints, RTD will execute U before V, and V before W. But then local propagation ensures that $V - W \leq -4$ and $U - V \leq -1$ will necessarily hold. Next, if RTD executes X before U, then $X \leq U < U + 2$ ensures that $X - U \leq 2$ will hold. But if RTD executes U before X, then local propagation ensures that $X - U \leq 2$ will hold. Similar remarks apply to the constraint $Y - X \leq 6$. Since the RTD algorithm necessarily satisfies each edge constraint along the path from W to Y, it necessarily satisfies the *path* constraint $Y - W \leq 3$.

2.2.2. eSTNU dispatchability

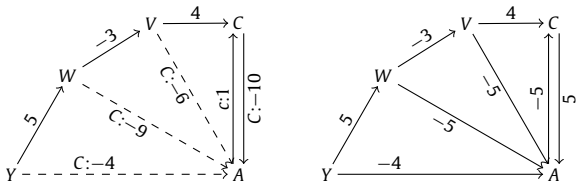
Morris [24] generalized the notion of dispatchability from STNs to *extended* STNUs (eSTNUs), which are STNUs along with any *generated* UC edges (e.g., the dotted UC edges in Fig. 2b). Recall that a generated UC edge such as $(W, C{:}{-5}, A)$ represents a *conditional* constraint, called a *wait:* as long as C remains unexecuted, W must *wait* until 5 after A. His main result builds on the notion of a *projection* for an eSTNU, which is an *STN* that results from choosing fixed durations for the contingent links.

Projection. Let $\mathcal{G} = (\mathcal{T}_X \cup \mathcal{T}_C, \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u \cup \mathcal{E}_{u_g})$ be any eSTNU graph where $\mathcal{E}_u$ contains the *original* UC edges, and $\mathcal{E}_{u_g}$ the *generated* UC edges (i.e., waits). A *situation* for $\mathcal{G}$ is any function, $\omega\colon \mathcal{T}_C \to \mathbb{R}$, that specifies fixed durations for the contingent links. In particular, for each $(A, x, y, C) \in \mathcal{L}$, $\omega(C) \in [x, y]$. (We'll write ω_c instead of $\omega(C)$.) The *projection* of $\mathcal{G}$ in the situation ω is the STN $(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_\ell^\omega \cup \mathcal{E}_u^\omega \cup \mathcal{E}_{u_g}^\omega)$, where for each $(A, x, y, C) \in \mathcal{L}$:

$$\begin{array}{lll} \text{(LC)} & (A, c{:}x, C) \in \mathcal{E}_\ell & \text{iff} \quad (A, \omega_c, C) \in \mathcal{E}_\ell^\omega; \\ \text{(UC)} & (C, C{:}{-}y, A) \in \mathcal{E}_u & \text{iff} \quad (C, -\omega_c, A) \in \mathcal{E}_u^\omega; \text{ and} \\ \text{(Waits)} & (V, C{:}{-}v, A) \in \mathcal{E}_u^\omega & \text{iff} \quad (V, -\delta, A) \in \mathcal{E}_{u_g}^\omega, \text{ where } \delta = \min\{v, \omega_c\}. \end{array}$$

The edges in $\mathcal{E}_\ell^\omega$ and $\mathcal{E}_u^\omega$ fix each contingent duration $C - A$ to the value ω_c. The edges in $\mathcal{E}_{u_g}^\omega$ depend on the relationship between the wait times and ω_c. For example, Fig. 6 shows a projection where $\omega_c = 5$ for the contingent link $(A, 1, 10, C)$. The waits for V and W expire when C executes at 5, and thus appear as unconditional lower bounds of 5 in the projection, while the wait for Y expires at 4, before C executes, and thus appears as an unconditional lower bound of 4 in the projection.

The following theorem, due to Morris [24], characterizes the dispatchability of an eSTNU in terms of its STN projections.



(a) Original eSTNU (b) STN Projection ($\omega_c = 5$)

Fig. 6. Projecting an eSTNU onto the situation where $\omega_c = 5$.

Theorem 2 *([24]). An eSTNU is dispatchable if and only if all of its* STN projections *are dispatchable (as STNs).*[3]

Morris augmented his 2014 DC-checking algorithm to not only compute all intermediate (dotted) edges, both ordinary and upper-case, but also *insert* them into the graph. He then proved that, for DC inputs, the algorithm, which still runs in $O(n^3)$ time, necessarily outputs an equivalent dispatchable eSTNU. In contrast, the RUL$^-$ DC-checking algorithm offers no such guarantee. *Therefore, Morris' $O(n^3)$ algorithm is the fastest so far for transforming an STNU into dispatchable form.* (Recall that the RUL$^-$ DC-checking algorithm has no guarantee of returning a dispatchable network. It typically does not return a dispatchable network.)

3. A faster dispatchability algorithm for STNUs

This section presents a novel algorithm, called FD$_{\text{STNU}}$, where FD stands for *Fast Dispatch.* Like Morris' 2014 algorithm, our FD$_{\text{STNU}}$ algorithm first checks the dynamic controllability of the input STNU and then, for DC instances, converts it into an equivalent dispatchable eSTNU. Its worst-case time complexity is $O(mn + kn^2 + n^2 \log n)$, where k is the number of contingent links and m is the number of ordinary edges. This complexity is better than the $O(n^3)$-time complexity of the Morris 2014 algorithm, especially for sparse graphs (e.g., when $k \ll n$ and $m \ll n^2$). Since the Morris 2014 algorithm is the only other algorithm with a dispatchability guarantee for DC STNUs, FD$_{\text{STNU}}$ is now the fastest such algorithm. The rest of this section describes the FD$_{\text{STNU}}$ algorithm in detail and then proves its correctness.

The FD$_{\text{STNU}}$ algorithm has three phases, as follows.

(1) It runs a variant of the RUL$^-$ algorithm, augmented to generate and insert new UC edges that correspond to *wait constraints.*
(2) It processes each *LC* edge, propagating *forward* along LO-edges, looking for opportunities to generate ordinary bypass edges.
(3) It runs the FDM algorithm [29] on the subgraph of ordinary edges, which is an STN, to convert it into a *dispatchable* STN subgraph.

To motivate the different phases of the FD$_{\text{STNU}}$ algorithm, consider the sample STNU shown in Fig. 7a, which happens to be dynamically controllable. Fig. 7b shows the same STNU together with the edges generated by the *original version* of the RUL$^-$ algorithm. In particular, the brown, dashed edges are obtained by applying the U$_{\text{nlp}}$ rule, and the cyan, dotted edge is obtained by applying the R rule. Even with these new edges, the STNU is not dispatchable, as demonstrated, for example, in Fig. 7c, which shows the STN projection in the situation where $\omega_c = 4$. Notice that this projection has paths from A to W, and from C to Y, but no corresponding vee-paths. Hence, for example, an STN dispatcher could inadvertently execute A, C, and Y at times 10, 14, and 16, respectively, before discovering (too late) that no value for X could satisfy $18 = Y + 2 \leq X \leq C + 3 = 17$. Similarly, it could execute A and Y at 0 and 1, respectively, before discovering that C could not satisfy $4 = 4 + A \leq C \leq Y + 1 = 2$. Or it could execute A and W at 0, before discovering that C cannot satisfy $7 = 7 + W \leq C \leq A + 4 = 4$.

In contrast, Fig. 7d shows how FD$_{\text{STNU}}$ processes the same DC STNU. The first phase applies the RUL$^-$ algorithm, propagating backward from the original UC edge $(C, C{:}-10, A)$; but, instead of using the U$_{\text{nlp}}$ rule to generate the edges $(Y, -1, A)$ and $(X, -1, A)$, it uses the UC rule (from Table 2) to generate the (brown, dashed) wait edges $(Y, C{:}-9, A)$ and $(X, C{:}-11, A)$. Next, the second phase propagates *forward* from the LC edge $(A, c{:}1, C)$, using the LC rule (from Table 2) to generate the (teal, dash-dotted) edge $(A, -6, W)$. Finally, the third phase runs the FDM algorithm on the *ordinary* STN subgraph, inserting the (red, dotted) edges $(C, 1, Y)$ and $(Y, -6, W)$. Fig. 7e shows the STN projection for the situation where $\omega_c = 4$. It is easy to check that this projection is vee-path-complete and, hence, by Theorem 1, dispatchable. It can similarly be shown that every STN projection of the eSTNU in Fig. 7d is dispatchable (as an STN), and hence that the eSTNU output by the FD$_{\text{STNU}}$ algorithm is dispatchable (as an eSTNU).

The next paragraphs provide more detail about the three phases of the new FD$_{\text{STNU}}$ algorithm.

[3] Equivalently, Morris [24] first proved the result and then *defined* dispatchability for eSTNUs in terms of the dispatchability of the STN projections.

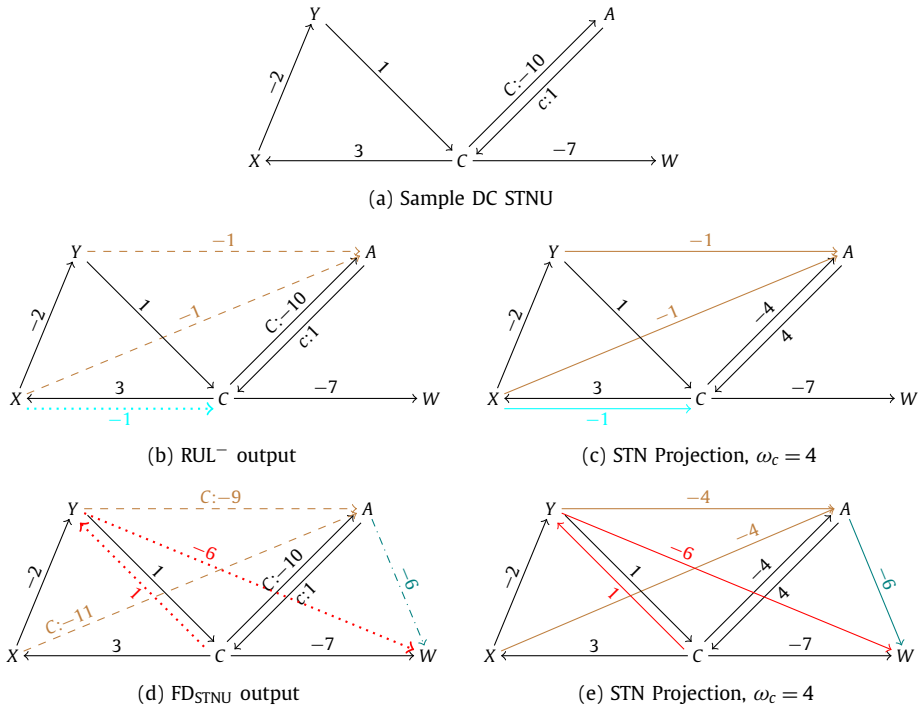


Fig. 7. Comparing RUL⁻ and FD_{STNU} on a DC STNU.

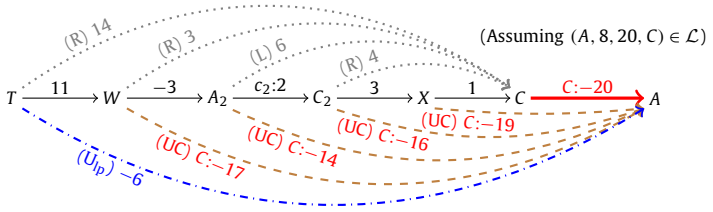


Fig. 8. Edges generated by Phase 1 of FD_{STNU}.

Phase 1: Modified RUL⁻. This phase involves running a slightly modified version of the RUL⁻ algorithm. (If the input STNU is not DC, then Phase 1 will return $\perp$.) Hunsberger and Posenato [18] recently showed that for DC checking, it is not necessary to *insert* the (dotted) intermediate edges generated by the R and L rules into the STNU graph (cf. Fig. 3); it suffices to merely keep track of the associated distances. They also showed that the U_{nlp} rule, and all of the edges it would generate, can be completely avoided in exchange for doing a limited amount of forward propagation from LC edges in certain (rare) circumstances, discussed in a moment. By dramatically reducing the number of edges that the algorithm inserts into the graph, the many instances of Dijkstra-like propagation run much faster. They showed that their algorithm, called RUL2021, runs up to an order of magnitude faster on a variety of STNU benchmarks, although it has the same worst-case complexity.

Fig. 8 illustrates how Phase 1 exploits the distance information associated with the (dotted) intermediate edges to generate and insert new UC edges that correspond to wait constraints, shown as brown and dashed, with red labels. In particular, for each dotted edge, which Phase 1 computes but does not insert into the graph, the UC rule (from Table 2) is applied to combine that edge with the original (thick red) UC edge to generate, and insert, a new UC edge. For example, in Fig. 8, the UC rule applied to the dotted edge $(C_2, 4, C)$ and the original UC edge $(C, C{:}-20, A)$ generates the new UC edge $(C_2, C{:}-16, A)$. Along with the new UC edges, Phase 1 also inserts ordinary bypass edges, such as the blue dash-dotted edge in Fig. 8, which RUL2021, like RUL⁻, computes using the U_{lp} rule.

As mentioned earlier, in exchange for *not* inserting the intermediate (dotted) edges during backpropagation, RUL2021 does a *limited* amount of *forward* propagation. (This forward propagation is only included for the purpose of DC checking; it does *not* generate any new edges.) The forward propagation is triggered in the (rare) cases where back-propagation from a UC edge $(C, C{:}-y, A)$ encounters a cycle, from C back to C, whose length is less than $\Delta_C = y - x$. Such a cycle has less flexibility than the contingent link; therefore, if any of its timepoints are constrained to occur *before* C, the network cannot be DC. To check this, RUL2021 propagates *forward* from C along LO-edges, keeping track of the shortest path-lengths it encounters. (In addition, it only visits timepoints that belong to the triggering cycle.) Fig. 9 shows two similar-looking, but critically different examples where this kind of forward propagation has been triggered because the preceding

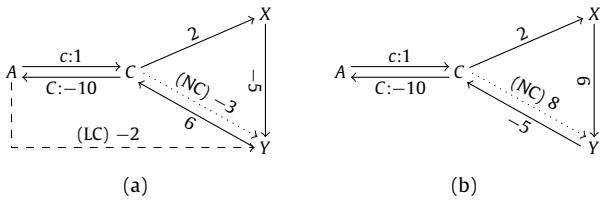


Fig. 9. Two cases of a cycle (C, X, Y, C) of length $3 < \Delta_C = 10 - 1 = 9$ triggering forward propagation by RUL2021.

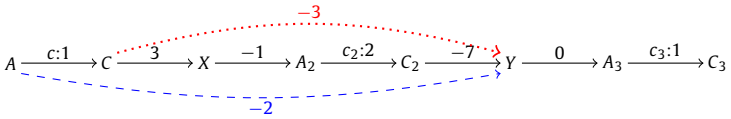


Fig. 10. Sample forward propagation during Phase 2 of the FD_{STNU} algorithm.

back-propagation from $(C, C{:}-10, A)$ encountered a cycle (C, X, Y, C) of length $3 < \Delta_C$. In the lefthand graph, forward propagation from C to X to Y reveals a negative-length path from C to Y which, for illustration, is represented in the figure by the dotted edge $(C, -3, Y)$. At this point, RUL2021 immediately stops because the STNU cannot be DC. (To see why, note that the LC rule could then generate the dashed bypass edge $(A, -2, Y)$, which would create a negative cycle, from A to Y to C to A, in the OU-graph.) In the righthand graph, no negative-length path emanating from C in the LO-graph is encountered. As a result, RUL2021 can resume its processing of UC edges, picking up wherever it had left off.[4]

Finally, to greatly simplify the proof of correctness for the new algorithm, Phase 1 uses a slightly relaxed version of the R rule from Table 3 in which the constraint $Q \in \mathcal{T}_X$ is removed (i.e., Phase 1 allows the middle timepoint in the R rule to be any timepoint). This relaxed version of the R rule is still sound, since it is a version of the sound NC rule from Table 2 with stricter applicability conditions.

Phase 1 can be done in $O(mn + k^2n + kn \log n)$, the worst case complexity of RUL2021 algorithm, where m is the number of edges, n the number of nodes, and $k < n$ the number of contingent links.

Phase 2: Forward propagation. Because Phase 1 processes UC edges by propagating backward along LO-edges, the edges it generates to bypass UC edges may also inadvertently bypass some LC edges. For example, the LC edge $(A_2, c_2{:}2, C_2)$ is bypassed in Fig. 8 by the wait edge $(A_2, C{:}-14, A)$. However, because such back-propagation only starts from UC edges, it is not guaranteed to generate all possible bypass edges for LC edges. For example, for the STNU from Fig. 7a, neither RUL^- nor RUL2021 would generate the edge $(A, -6, W)$, which bypasses the LC edge $(A, c{:}1, C)$.

Since, as illustrated in Fig. 7d-e, these kinds of bypass edges are needed to ensure dispatchability, Phase 2 of FD_{STNU} carries out forward propagations from each of the k LC edges, along paths in the LO-graph, looking for opportunities to generate ordinary edges that bypass the LC edges. When processing an LC edge $(A, c{:}x, C)$, FD_{STNU} propagates forward from the contingent timepoint C, keeping track of the lengths of the shortest paths emanating from C in the LO-graph. Whenever the forward propagation reveals a negative-length path emanating from C, the LC rule (from Table 2) is used to generate an ordinary bypass edge (e.g., the edge $(A, -6, W)$ in Fig. 7d). Although similar to the limited forward propagation done during Phase 1, the goal here is not to check for negative cycles in the OU-graph—that has already been done by Phase 1—but instead to generate bypass edges that will contribute to making the output of FD_{STNU} a dispatchable eSTNU.[5]

Fig. 10 illustrates the forward propagation during Phase 2. In the figure, forward propagation from the contingent timepoint C continues until the *first time* the length of the path turns negative—in this case, when the path of length -3 from C to Y is encountered. Morris [23] called such paths *extension sub-paths.* Because the extension sub-paths explored during Phase 2 contain only LO-edges, Morris' analysis ensures that the rules from Table 2 can always be applied to generate the corresponding bypass edges. In addition, the bypass edges generated by Phase 2 will invariably be ordinary edges. For example, in Fig. 10, several applications of the rules from Table 2 can be used to generate the (red, dotted) edge $(C, -3, Y)$; and then the LC rule (also from Table 2) can be used to generate the ordinary (blue, dashed) bypass edge $(A, -2, Y)$. Crucially, because the rules are length preserving, and always generate ordinary edges, there is no need for Phase 2 to actually *apply* the rules. Instead, Phase 2 need only compute the lengths of shortest paths emanating from C in the LO-graph and, whenever it encounters a negative-length path, immediately generate the ordinary bypass edge. Once a bypass edge such as $(A, -2, Y)$ is generated, no further propagation past Y is carried out.

Since Phase 1 ends with an updated potential function based on the LO-edges, Phase 2 can use that potential function to guide its forward propagations in the LO-graph. Furthermore, because the rules that it uses to generate bypass edges are length preserving, and the paths it explores do not involve any UC edges, inserting those new ordinary edges into the

[4] The RUL^- algorithm deals with the scenarios in Fig. 9 quite differently. For each graph, it uses the U_{nlp} rule to generate the edge $(Y, -1, A)$ which, on the left, creates a negative cycle, from A to C to X to Y to A, in the LO-graph, but not on the right.

[5] The forward propagations done during Phase 1 are insufficient for this purpose because they are triggered only when certain cycles are encountered, and they only explore nodes belonging to the triggering cycles.

network cannot disturb the potential function. As a result, Phase 2 never needs to update the potential function. In addition, for efficiency, all bypass edges generated during Phase 2 are not actually inserted into the graph until the very end of the phase. Therefore, since Phase 1 ends with at most $m^* = m + kn$ edges, and each of the k runs of Dijkstra can be done in $O(m^* + n \log n)$ time, Phase 2 can be done in $O(k(m+kn) + kn \log n) = O(km + k^2 n + kn \log n)$ time overall.

Phase 3: FDM. The pre-existing *fastDispatchMin* (FDM) algorithm takes an STN as input and generates as output an equivalent minimal dispatchable STN. It ensures that, whenever there is a path from any X to any Y, there is necessarily a shortest path from X to Y that is also a *vee-path.* Phase 3 of FD_{STNU} applies the FDM algorithm to the STN subgraph of the eSTNU that comprises only the ordinary edges, transforming it into a dispatchable STN subgraph. In the process, some new ordinary edges may be inserted into the graph (e.g., the red edges in Fig. 7c), while others may be removed. For example, if the edge $(X, -1, C)$ had been in the ordinary subgraph of Fig. 7c, then FDM would have removed it, given the alternative vee-path from X to Y to C of length -1.

Since Phases 1 and 2 each insert at most kn new edges, Phase 3 runs in $O((m+kn)n + n^2 \log n) = O(mn + kn^2 + n^2 \log n)$ time.

Complexity of FD_{STNU}. The overall complexity of the FD_{STNU} algorithm derives from the complexities of its three phases, as follows:

$$\begin{aligned}&\text{Phase 1} + \text{Phase 2} + \text{Phase 3}\\ &= O(mn + k^2 n + kn \log n) + O(km + k^2 n + kn \log n) + O(mn + kn^2 + n^2 \log n)\\ &= O(mn + kn^2 + n^2 \log n)\end{aligned}$$

Theorem 3. *Let $\mathcal{G}$ be any DC STNU. Then the eSTNU obtained by running the FD_{STNU} algorithm on $\mathcal{G}$ is necessarily a dispatchable eSTNU.*

Proof. The following is a sketch of the proof. A much more detailed proof is given in the Technical Appendix.

Let $\mathcal{G}^*$ be the eSTNU output by FD_{STNU}. By Theorem 2, $\mathcal{G}^*$ is dispatchable if and only if all of its STN projections are dispatchable (as STNs). By Theorem 1, each STN projection is dispatchable if and only if it is vee-path-complete. Therefore, let ω be any situation, $\mathcal{G}^\omega$ the corresponding STN projection of $\mathcal{G}^*$, and $\mathcal{P}^\omega_{xy}$ any path in $\mathcal{G}^\omega$ from some X to some Y. It suffices to prove that there is a shortest path from X to Y in $\mathcal{G}^\omega$ that is also a vee-path. With no loss of generality, assume $\mathcal{P}^\omega_{xy}$ is a shortest path in $\mathcal{G}^\omega$.

Let $\mathcal{P}^*_{xy}$ be a path in $\mathcal{G}^*$ that projects onto $\mathcal{P}^\omega_{xy}$ in the situation ω. The rest of the proof has three parts. First, process the upper-case edges in $\mathcal{P}^*_{xy}$ from left to right, showing that each can be bypassed (by a path in $\mathcal{G}^*$) either by an ordinary edge or a path consisting of zero or more negative ordinary edges followed by a UC edge (a so-called nU-subpath). After processing all of the UC edges, the result is a path $\mathcal{P}'_{xy}$ from X to Y in $\mathcal{G}^*$ that consists of zero or more nU-subpaths, followed by zero or more LO-edges. Crucially, all of the edges in the prefix consisting of the nU-subpaths are negative, and hence project onto a path with only negative edges.

The second part involves processing the LC edges in the LO remainder of $\mathcal{P}'_{xy}$ from right to left, showing that, for each, either (1) it can be bypassed by an ordinary edge (in $\mathcal{G}^*$), or (2) the subpath of ordinary edges following it can be replaced by all non-negative ordinary edges (in $\mathcal{G}^*$). An LC edge followed by zero or more non-negative ordinary edges is henceforth called an Lnn-subpath. Crucially, all of the edges in an Lnn-subpath are non-negative, and hence project onto a path that comprises only non-negative edges. After processing all of the LC edges, the result is a path $\mathcal{P}''_{xy}$ from X to Y in $\mathcal{G}^*$ that is the concatenation of (1) zero or more nU-subpaths, (2) an ordinary path (i.e., a path consisting of only ordinary edges), and (3) zero or more Lnn-subpaths.

The third part notes that since Phase 3 of FD_{STNU} ensures that the subgraph of ordinary edges is STN-dispatchable, the middle portion of $\mathcal{P}''_{xy}$ can be replaced, if necessary, by an ordinary vee-path (in $\mathcal{G}^*$). This ensures that the resulting path, $\mathcal{P}^\dagger_{xy}$, projects onto a vee-path in $\mathcal{G}^\omega$. Finally, the Technical Appendix confirms that the length of the projection of $\mathcal{P}^\dagger_{xy}$ is no more than the length of the original path $\mathcal{P}^\omega_{xy}$. □

Theorem 4. *If the input $\mathcal{G}$ is DC, then the FD_{STNU} algorithm necessarily outputs an equivalent eSTNU.*

Proof. All rules used by FD_{STNU} have been proven to be sound by the researchers that introduced them [27,23,5]. As a result, all edges generated during Phases 1 and 2 must be respected by any dynamic execution strategy for the eSTNU. Finally, Phase 3 replaces the ordinary subgraph (an STN) with an equivalent ordinary subgraph (another STN). Therefore, all of the edges in the eSTNU output by FD_{STNU} must be satisfied by any dynamic strategy for the input STNU, and vice-versa. □

3.1. Pseudocode

This section presents pseudocode for the *new* procedures used by the FD_{STNU} algorithm. Pseudocode for `RUL2021` and `fastDispatchMin` is available elsewhere [18,29].

Algorithm 1: The FD_{STNU} algorithm.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, an STNU graph.
Output: A dispatchable eSTNU graph, if $\mathcal{G}$ is DC; $\perp$, otherwise.
// Phase 1: checks DC and, if successful, generates a solution, f, for the LO-graph and *distToC*, a map $C \mapsto [d_1, \ldots, d_n]$, where for each $X_i \in \mathcal{T}$, d_i is the minimum distance from X_i to C in the LO-graph.
1 $(\mathcal{G}_1 = (\mathcal{T}, \mathcal{E}_1), f, distToC)$:= `RUL2021`$(\mathcal{G})$
2 **if** f == $\{\}$ **then return** $\perp$
// Below, insert all waits accumulated by modified `RUL2021`
3 $accum := \{\}$
4 **foreach** *contingent link* $(A, x, y, C) \in \mathcal{E}_1$ **do**
5 $\quad accum := accum \cup$ `genWaitEdges`$(\mathcal{G}_1, (A, x, y, C), distToC(C))$ // cf. Alg. 2
6 $\mathcal{G}_{1b}$:= $(\mathcal{T}, \mathcal{E}_{1b} = \mathcal{E}_1 \cup accum)$
// Phase 2: inserts ordinary edges that bypass LC edges
7 $accum := \{\}$
8 **foreach** *contingent link* $(A, x, y, C) \in \mathcal{E}_{1b}$ **do**
9 $\quad accum := accum \cup$ `fwdProp`$(\mathcal{G}_{1b}, (A, c{:}x, C), f)$ // cf. Alg. 3
10 $\mathcal{G}_2$:= $(\mathcal{T}, \mathcal{E}_2 = \mathcal{E}_{1b} \cup accum)$
// Phase 3: makes ordinary subgraph STN-dispatchable
11 $\mathcal{G}_3$:= `fastDispatchMin`$(\mathcal{G}_2, f)$
12 **return** $\mathcal{G}_3$

The FD$_{\text{STNU}}$ algorithm. Algorithm 1 gives the high-level structure of the FD_{STNU} algorithm. If the input STNU $\mathcal{G}$ is DC, FD_{STNU} returns an equivalent, dispatchable eSTNU; otherwise, it returns $\perp$.

Phase 1 of FD_{STNU} is realized in two steps. In the first step, at Line 1, FD_{STNU} calls the `RUL2021` algorithm to check whether $\mathcal{G}$ is dynamically controllable (DC). If it is, `RUL2021` outputs the STNU $\mathcal{G}_1 = (\mathcal{T}, \mathcal{E}_1)$, where $\mathcal{E}_1$ contains all the edges of $\mathcal{E}$, together with ordinary edges that bypass UC edges. We modified `RUL2021` to also return (1) f, a solution/potential function for the LO-graph; and (2) *distToC*, a map that provides, for each contingent node C, an array of the minimum distances from each node X to C in the LO-graph. These are minor modifications since these data are already computed by `RUL2021`. In cases where $\mathcal{G}$ is not DC, then f is empty and the check in Line 2 makes FD_{STNU} halt, returning $\perp$.

The second step of Phase 1, at Lines 3 to 6, involves calling `genWaitEdges` (Algorithm 2, discussed below), once for each contingent link, to compute and insert the wait constraints derived from the distances stored in *distToC* (cf. the brown dashed edges in Fig. 8). The resulting eSTNU is called $\mathcal{G}_{1b}$.

Phase 2 of FD_{STNU} is realized in Lines 7 to 10. For each LC edge e, FD_{STNU} calls `fwdProp` (Algorithm 3, discussed below), which propagates forward from e to generate ordinary edges that bypass e. It uses the potential function f to enable a Dijkstra-like propagation along LO-edges. $\mathcal{G}_2$ is the eSTNU graph obtained by inserting the ordinary bypass edges into $\mathcal{G}_{1b}$.

Finally, Phase 3, at Line 11, calls `fastDispatchMin` on $\mathcal{G}_2$. (We modified `fastDispatchMin` to take the eSTNU $\mathcal{G}_2$ as input, but it only operates on the ordinary STN subgraph.) The eSTNU $\mathcal{G}_3$ is obtained by replacing the ordinary edges in $\mathcal{G}_2$ by the equivalent ordinary subgraph computed by `fastDispatchMin`. $\mathcal{G}_3$ is the dispatchable eSTNU output by FD_{STNU}.

The `genWaitEdges` algorithm. Algorithm 2 gives pseudocode of the algorithm `genWaitEdges`. Given a DC STNU $\mathcal{G}$, and a contingent link (A, x, y, C), `genWaitEdges` returns the set of wait edges (e.g., the brown dashed edges with red values shown in Fig. 8) that result from applying the UC rule to the UC edge $(C, C{:}-y, A)$ and each of the *intermediate* edges (e.g., the dotted edges in Fig. 8) that are computed, but not inserted, by `RUL2021`. To speed up `genWaitEdges`, the distance from each X to C in the LO-graph, computed by `RUL2021`, is provided in the vector $dist = [d_1, \ldots, d_n]$, which derives from $distToC[C]$ in Algorithm 1.

Algorithm 2: The `genWaitEdges` algorithm.

Input: $\mathcal{G}$, a DC STNU graph; (A, x, y, C), a contingent link; $dist = [d_1, \ldots, d_n]$, where for each i, d_i is the minimum distance from X_i to C in the LO-graph $\mathcal{G}_{\ell o}$.
Output: The set of UC edges/waits obtained by applying the UC rule to the (dotted) edges corresponding to distances in *dist* and the UC edge $(A, C{:}-y, C)$.
1 $newWaits := \{\}$ // For accumulating generated UC/wait edges
2 $\Delta_C := y - x$
3 **foreach** $X_i \in \mathcal{T}$ **do**
4 $\quad$ **if** $X_i \not\equiv C$ **then** // I.e., X_i and C are distinct time-points
5 $\quad\quad \delta_{xc} := dist[X_i] = d_i$
6 $\quad\quad$ **if** $\delta_{xc} < \Delta_C$ **then** // Applicability condition
7 $\quad\quad\quad newWaits := newWaits \cup (X_i, C{:}(\delta_{xc} - y), A)$ // Generate UC edge
8 **return** *newWaits*

For each X_i, δ_{xc} (at Line 5) holds the distance from X to C in the LO-graph. If $\delta_{xc} \geq \Delta_C$, then `RUL2021` would have used the U_{lp} rule to generate an ordinary bypass edge (e.g., the blue, dashed edge in Fig. 8). Otherwise, $\delta_{xc} < \Delta_C$ and `genWaitEdges` generates the UC edge, $(X_i, C{:}\delta_{xc} - y, A)$ (Lines 6 to 7). The set of UC edges is returned at Line 8.

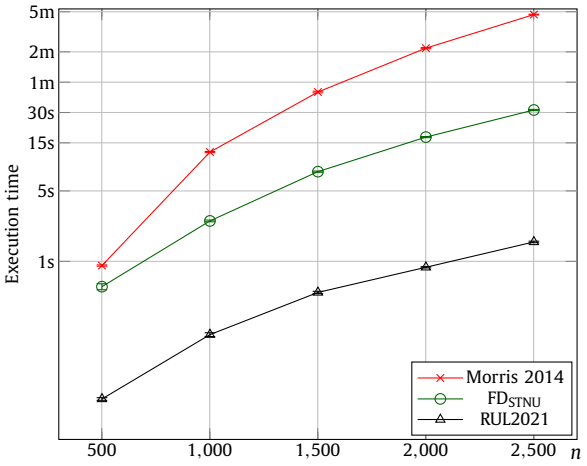


Fig. 11. Execution time vs number of nodes, n.

The `fwdProp` ***algorithm***. Pseudocode for `fwdProp` is given as Algorithm 3. For a given contingent link (A, x, y, C), it does a forward propagation along edges in the LO-graph emanating from C, aiming to generate ordinary edges that bypass the LC edge $(A, c{:}x, C)$. It uses the potential function f to re-weight edges in the LO-graph to be non-negative, thereby enabling a Dijkstra-like traversal of paths. The key for each X in the priority queue is $d(C, X) - f(X)$, where $d(C, X)$ is the distance from C to X in the LO-graph. Whenever some $d(C, X) < 0$ (Line 7), a bypass edge is generated. Otherwise, forward propagation along edges emanating from X continues (Lines 10 to 16).

Algorithm 3: The `fwdProp` algorithm.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, a DC eSTNU graph; (A, x, y, C), a contingent link; f, a solution for the LO-graph $\mathcal{G}_{\ell o}$.
Output: A set of ordinary edges that bypass the LC edge $(A, c{:}x, C)$.

```
1  Q := new empty priority queue
   // For each X, key_X = d(C, X) − f(X), where d(C, X) is the dist. from C to X in G_ℓo.
2  Q.insert(C, −f(C))                                   // key_C = d(C, C) − f(C) = 0 − f(C) = −f(C)
3  accum := {}
4  while ¬Q.empty() do
5      (X, key_X) := Q.extractMin()
6      d(C, X) := key_X + f(X)
7      if d(C, X) < 0 then                              // Signal to generate a bypass edge
8          accum := accum ∪ (A, x + d(C, X), X)         // Accumulate bypass edge
9      else                                             // Otherwise, propagate forward
10         foreach (X, δ_XY, Y) ∈ E_o ∪ E_ℓ do
11             if Y ≢ C and Y ≢ A then                  // I.e., Y is distinct from A and C
12                 key_Y := d(C, X) + δ_XY − f(Y)
13                 if Q.state(Y) == notYetInQ then
14                     Q.insert(Y, key_Y)
15                 else if Q.state(Y) == inQ then
16                     Q.decreaseKey(Y, key_Y)
17 return accum
```

4. Experimental evaluation

We evaluated the performance of the FD_{STNU} algorithm against two pre-existing algorithms: Morris 2014, the previous state-of-the-art for computing dispatchable eSTNUs; and RUL2021, which only does DC checking, to see how much the move from DC checking to computing dispatchable network costs. All algorithms were implemented in Java and run on a JVM 17 with 8 GB of heap memory on a Mac OS X box with an Intel(R) Quad-Core Intel Core i7@2.6GHz. Our implementations are available in the CSTNU Tool [33]. Summaries of our tests and results are given below.

We used two of the STNU benchmarks published by Posenato [32] and already used for a previous work [18].

For each $n \in \{500, 1000, 1500, 2000, 2500\}$, the first benchmark (called *Test 1*) contains 200 randomly generated DC STNUs, each having n nodes, $n/10$ contingent links, and around 6 incident edges for each node (for a total of $m \approx 3n$ edges).

Fig. 11 displays the average execution times; each point represents the average execution time for an algorithm on the first 100 instances of the given size. The error bars, which show 95% confidence intervals, are not visible because standard

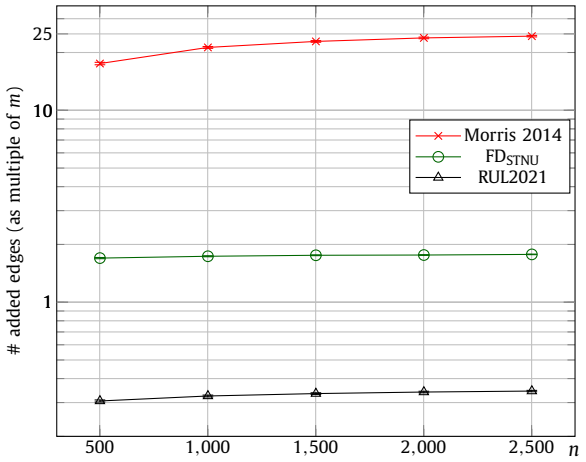


Fig. 12. Number of added edges vs number of nodes, n.

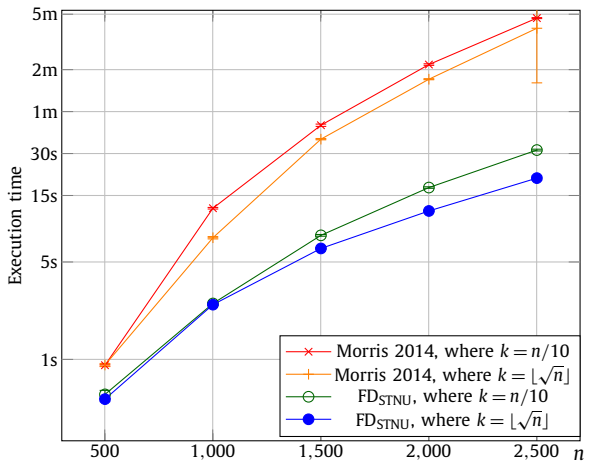


Fig. 13. Execution time vs. number of nodes, n, comparing performance on instances in which, k, the number of contingent links is $n/10$ or $\lfloor\sqrt{n}\rfloor$.

deviation values are around 2% of average execution times. These results confirm that FD_{STNU} outperforms the previous state-of-the-art, Morris 2014, for computing dispatchable eSTNUs. In addition, it highlights that computing a dispatchable network is roughly an order of magnitude more expensive than DC checking by RUL2021.

Fig. 12 plots the number of edges inserted by each algorithm as a multiple of the number of edges in the input graph, using a log scale. It shows that FD_{STNU} generates a dispatchable network adding, on average, fewer than $1.8m$ new edges, while Morris 2014 averages around $20m$ new edges.

The second benchmark was obtained from the first by reducing the number of contingent links, k, from $n/10$ to $\lfloor\sqrt{n}\rfloor$. ("Extra" contingent links were converted into ordinary constraints.) Fig. 13 displays the average execution times, each point representing the average execution time for an algorithm on 100 instances of the given size. The error bars, which show 95% confidence intervals, are not visible because standard deviation values are around 2% of average added edges. These results confirm that the FD_{STNU} algorithm performs better on instances where $k = \lfloor\sqrt{n}\rfloor$, with a smaller execution-time grow-rate, as compared to its performance on *Test 1,* with $k = n/10$. In addition, although the Morris 2014 algorithm also performs better on instances where $k = \lfloor\sqrt{n}\rfloor$, as compared to its performance on instances where $k = n/10$, its execution-time grow-rate appears to be the same across the two benchmarks.

5. Conclusion

This paper introduced a new algorithm, FD_{STNU}, that converts DC STNUs into dispatchable form. Its complexity is $O(mn + kn^2 + n^2 \log n)$, which is significantly better than the previous best, Morris' $O(n^3)$ algorithm, when networks are sparse. For example, if $m = O(n \log n)$ and $k = O(\log n)$, then FD_{STNU} algorithm runs in $O(n^2 \log n) \ll O(n^3)$ time. The paper proved the algorithm's correctness and presented an empirical evaluation demonstrating its better performance on existing STNU benchmarks.

Future work will focus on computing a *minimal* dispatchable network for DC STNUs (i.e., an equivalent, dispatchable STNU having the fewest edges). No existing algorithm for computing minimal dispatchable STNUs currently exists.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Benchmarks are available in a free-access repository [32].

Acknowledgments

This work was partially supported by the National Science Foundation [Grant No. 1909739].

Appendix A. Proofs

A.1. Notation and preliminary results

In this section, we shall continue to use upper-case letters such as A, B, C, U, V, W, X and Y to denote time-points. However, the upper-case letter E, often augmented by subscripts, shall be reserved for denoting upper-case edges, while the lower-case letter e shall be used to denote lower-case edges. In addition, the following lemma uses F to denote any arbitrary edge.

Definition 1 *(Projection notation).* For any (LC, UC or ord) edge F in an eSTNU graph $\mathcal{G}$, and any situation ω, let $\text{Pr}_\omega(F)$ denote the projection of F in the situation ω, and let $\|F\|_\omega = |\text{Pr}_\omega(F)|$ denote the length of its projection. Similarly, for any path $\mathcal{P}$ in $\mathcal{G}$, let $\text{Pr}_\omega(\mathcal{P})$ denote the path obtained by replacing each edge in $\mathcal{P}$ by its projection, and let $\|\mathcal{P}\|_\omega = |\text{Pr}_\omega(\mathcal{P})|$ denote the length of the projected path.

Lemma 1. *For any (LC, UC or ord) edge F in an eSTNU graph $\mathcal{G}$, and any situation ω, $|F| \leq \|F\|_\omega$ and, hence, for any path $\mathcal{P}$ in $\mathcal{G}$, $|\mathcal{P}| \leq \|\mathcal{P}\|_\omega$. Furthermore, $|F| < 0$ if and only if $\|F\|_\omega < 0$.*

Proof. If F is an ordinary edge $e_o = (U, \delta, V)$, then $|F| = \delta = \|F\|_\omega$. If F is an LC edge $(A, c{:}x, C)$, then $|F| = x \leq \omega_c = \|F\|_\omega$. If F is an original UC edge $(C, C{:}-y, A)$, then $|F| = -y \leq -\omega_c = \|F\|_\omega$. If F is a generated UC edge $(V, C{:}-v, A)$, then $|F| = -v \leq \max\{-\omega_c, -v\} = \|F\|_\omega$.

For the second claim, first suppose that $|F| \geq 0$. Then $0 \leq |F| \leq \|F\|_\omega$ implies that $\|F\|_\omega \geq 0$. On the other hand, if $|F| < 0$, then one of the following must hold: (1) F is an ord edge, in which case, $\|F\|_\omega = |F| < 0$; (2) F is an original UC edge, whence $\|F\|_\omega = -\omega_c < 0$; or (3) F is a generated UC edge, whence $\|F\|_\omega = \max\{-\omega, -v\} < 0$. (Generated UC edges necessarily satisfy $-v < 0$ given the applicability conditions for the U_{lp} rule.) □

A.2. Main result

Theorem 3. Let $\mathcal{G}$ be any DC STNU. Then the eSTNU obtained by running the FD_{STNU} algorithm on $\mathcal{G}$ is necessarily a dispatchable eSTNU that is equivalent to $\mathcal{G}$.

Proof. Let $\mathcal{G}$ be any DC STNU. Distinguish the changes made to $\mathcal{G}$ by FD_{STNU}, as follows. Let $\mathcal{G}_1$ be the eSTNU that results from running Phase 1 (i.e., modified RUL2021) on $\mathcal{G}$. Thus, $\mathcal{G}_1$ includes all edges from $\mathcal{G}$ together with the ord and UC bypass edges generated by Phase 1. Next, let $\mathcal{G}_2$ be the eSTNU that results from running Phase 2 (i.e., forward propagation from LC edges) on $\mathcal{G}_1$. Thus, $\mathcal{G}_2$ includes all edges from $\mathcal{G}_1$ together with any ord bypass edges generated by Phase 2. Next, let $\mathcal{G}_3$ be the eSTNU that results from running Phase 3 on $\mathcal{G}_2$ (i.e., running the FDM algorithm on the ord edges in $\mathcal{G}_2$). Thus, $\mathcal{G}_3$ includes all of the LC and UC edges from $\mathcal{G}_2$ together with an equivalent, but typically very different, set of ord edges. By construction, $\mathcal{G}_3$ is the output eSTNU generated by running the RUL2021 algorithm on $\mathcal{G}$. In addition, for any situation ω, the STN projection of $\mathcal{G}_3$ in that situation is denoted by $\mathcal{G}_\omega$.

Path notation. In this proof, a term such as $\mathcal{P}^i_{uv}$ will be used to denote a path from U to V in the graph $\mathcal{G}_i$. Different paths with the same endpoints and in the same graph are distinguished by alphabetic indices, primes or asterisks in the superscript. For example, $\mathcal{P}^2_{uv}, \mathcal{P}^{2a}_{uv}$ and $\mathcal{P}^{2'}_{uv}$ might denote three different paths from U to V in $\mathcal{G}_2$; and $\mathcal{P}^{\omega}_{xy}$ and $\mathcal{P}^{\omega^*}_{xy}$ might denote different paths from X to Y in the projection $\mathcal{G}_\omega$.

Given Theorem 2, it suffices to prove that every STN projection of the output eSTNU $\mathcal{G}_3$ is dispatchable. Toward that end, let ω be any situation and $\mathcal{G}_\omega$ the projection of $\mathcal{G}_3$ in that situation. Next, given Theorem 1, it suffices to show that $\mathcal{G}_\omega$ is

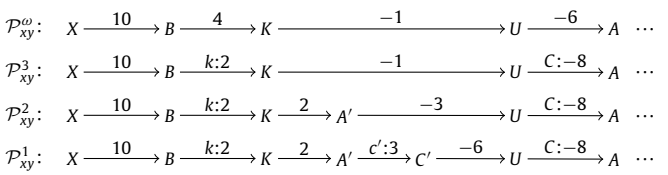


Fig. A.14. Illustrating paths in the proof of Theorem 3.

vee-path-complete. Toward that end, let $\mathcal{P}^{\omega}_{xy}$ be any shortest path in $\mathcal{G}_\omega$ from some X to some Y. It then suffices to show that there must exist a vee-path $\mathcal{P}^{\omega^*}_{xy}$ in $\mathcal{G}_\omega$ from X to Y such that $|\mathcal{P}^{\omega^*}_{xy}| \leq |\mathcal{P}^{\omega}_{xy}|$.

Now, by the definition of the projection $\mathcal{G}_\omega$, each edge in $\mathcal{P}^{\omega}_{xy}$ necessarily derives from a corresponding (LC, UC or ord) edge in $\mathcal{G}_3$. Therefore, as illustrated in Fig. A.14, there must exist a path $\mathcal{P}^{3}_{xy}$ in $\mathcal{G}_3$ whose edges project onto the corresponding edges in $\mathcal{P}^{\omega}_{xy}$ in the situation ω. Hence, $\Pr_\omega(\mathcal{P}^{3}_{xy}) = \mathcal{P}^{\omega}_{xy}$.

Next, note that because Phase 3 replaces the set of ord edges in $\mathcal{G}_2$ with an STN-equivalent set of ord edges in $\mathcal{G}_3$, it follows that each ordinary edge in $\mathcal{G}_3$ derives from either the same edge in $\mathcal{G}_2$ or an equivalent ordinary *path* in $\mathcal{G}_2$. Thus, there must exist a path $\mathcal{P}^{2}_{xy}$ in $\mathcal{G}_2$ obtained by replacing each ordinary edge in $\mathcal{P}^{3}_{xy}$ by an equivalent ord edge-or-path in $\mathcal{G}_2$. For example, in Fig. A.14, $(X, 10, B)$ appears in both $\mathcal{P}^{3}_{xy}$ and $\mathcal{P}^{2}_{xy}$, whereas $(K, -1, U)$ in $\mathcal{P}^{3}_{xy}$ corresponds to the equivalent two-edge path $(K, 2, A', -3, U)$ in $\mathcal{P}^{2}_{xy}$.

Now some of the ordinary edges in $\mathcal{P}^{2}_{xy}$ may be bypass edges generated during Phase 2, each of which derives from an LO-path in $\mathcal{G}_1$ comprising an LC edge e, followed by a negative-length LO-path $\mathcal{P}_e$. (The path $\mathcal{P}_e$ is called an *extension subpath* (ESP) for the LC edge e [23].) Therefore, there must exist a path $\mathcal{P}^{1}_{xy}$ in $\mathcal{G}_1$ obtained by replacing each ordinary bypass edge in $\mathcal{P}^{2}_{xy}$ by a corresponding LO-path from $\mathcal{G}_1$. For example, in Fig. A.14, the ordinary edge $(A', -3, U)$ in $\mathcal{P}^{2}_{xy}$ is replaced by the two-edge LO-path $(A', c'{:}3, C', -6, U)$ in $\mathcal{P}^{1}_{xy}$.

By construction, the UC edges in $\mathcal{P}^{3}_{xy}$ also appear in $\mathcal{P}^{2}_{xy}$ and $\mathcal{P}^{1}_{xy}$; and no other UC edges appear in any of these paths. As a result, these UC edges partition $\mathcal{P}^{3}_{xy}, \mathcal{P}^{2}_{xy}$ and $\mathcal{P}^{1}_{xy}$ in the same way. Similarly, the LC edges in $\mathcal{P}^{3}_{xy}$ also appear in $\mathcal{P}^{2}_{xy}$ and $\mathcal{P}^{1}_{xy}$. However, $\mathcal{P}^{1}_{xy}$ may also contain additional LC edges (e.g., $(A', c'{:}3, C')$ in Fig. A.14). In addition, since Phases 1, 2 and 3 are all length-preserving, $|\mathcal{P}^{1}_{xy}| = |\mathcal{P}^{2}_{xy}| = |\mathcal{P}^{3}_{xy}|$ and, by Lemma 1, $|\mathcal{P}^{3}_{xy}| \leq \|\mathcal{P}^{3}_{xy}\|_\omega = |\mathcal{P}^{\omega}_{xy}|$.

The rest of the proof has three sections. The first analyzes the UC edges; the second, the LC edges; and the third, the ord edges. Based on these analyses, $\mathcal{P}^{3}_{xy}$ is transformed into a path $\mathcal{P}^{3'}_{xy}$ whose projection is a vee-path whose length is at most $|\mathcal{P}^{\omega}_{xy}|$. In particular, $\mathcal{P}^{3'}_{xy}$ has a prefix consisting of negative OU-edges, a suffix consisting of non-negative LO-edges, and a middle section that is a vee-path of ordinary edges.

Analyzing UC edges in $\mathcal{P}_3$. Let $E_u = (U, C{:}-u, A)$ be the leftmost (i.e., first occurrence of a) UC edge in $\mathcal{P}^{3}_{xy}$, and let (A, x, y, C) be the corresponding contingent link. For example, in Fig. A.14, $E_u = (U, C{:}-8, A)$. Since E_u is the leftmost UC edge in $\mathcal{P}^{3}_{xy}$, it follows that the subpath $\mathcal{P}^{3}_{xu}$ from X to U comprises only LO-edges. It also follows that the corresponding subpaths from X to U in $\mathcal{P}^{2}_{xy}$ and $\mathcal{P}^{1}_{xy}$ also comprise only LO-edges, as illustrated in Fig. A.14. Finally, let $X = T_0, T_1, \ldots, T_f = U$ be the time-points preceding U in $\mathcal{P}^{1}_{xy}$. In Fig. A.14, this sequence of time-points would be X, B, K, A', C', U.

When Phase 1 processed the *original* UC edge $(C, C{:}-y, A)$, it did so by back-tracking from C along shortest LO-paths in $\mathcal{G}_1$, continuing as long as the path-length was less than Δ_C.[6] For each $T_i \in \{X = T_0, T_1, \ldots, T_f = U\}$, let $\mathcal{P}^{1}_{t_ic}$ denote a shortest LO-path from T_i to C in $\mathcal{G}_1$. (The $\mathcal{P}^{1}_{t_ic}$ paths need not be sub-paths of $\mathcal{P}^{1}_{xy}$.)

Case a.1: For some T_i, the shortest LO-path $\mathcal{P}^{1}_{t_ic}$ satisfies $|\mathcal{P}^{1}_{t_ic}| \geq \Delta_C$. In case there might be multiple such time-points, let T be the *rightmost* such T_i in $\mathcal{P}^{1}_{xy}$, and let $\mathcal{P}^{1}_{tc}$ denote a shortest LO-path from T to C in $\mathcal{G}_1$. Since $|\mathcal{P}^{1}_{tc}| \geq \Delta_C$, it follows that T cannot be U (i.e., T must *precede* U in $\mathcal{P}^{1}_{xy}$). (If E_u is a generated UC edge, then the shortest LO-path $\mathcal{P}^{1}_{uc}$ from U to C satisfies $|\mathcal{P}^{1}_{uc}| < \Delta_C$. Alternatively, if E_u is the original UC edge, then $|\mathcal{P}^{1}_{uc}| = |\mathcal{P}^{1}_{cc}| = 0 < \Delta_C$.) Furthermore, by the choice of T, for each T_j strictly *between* T and U in $\mathcal{P}^{1}_{xy}$, $|\mathcal{P}^{1}_{t_jc}| < \Delta_C$ and, hence, Phase 1 would have back-tracked along the edge preceding T_j in $\mathcal{P}^{1}_{xy}$. This implies that Phase 1 would necessarily have explored the entire path from U back to T. Therefore, Phase 1 must have processed T, at which point it must have generated an ordinary bypass edge $E_t = (T, -t, A)$, where $-t = |\mathcal{P}^{1}_{tc}| - y$.

[6] Phase 1 starts by back-tracking along LO-paths in $\mathcal{G}$ (i.e., the input STNU). But the processing of one UC edge might generate new ordinary edges, thereby increasing the pool of LO-edges available for subsequent back-tracking. In addition, should back-tracking from some C during Phase 1 encounter the activation time-point A' for some other contingent link, then Phase 1 interrupts its back-tracking from C until the UC edge $(C', C'{:}-y', A')$ has been fully processed. As a result, when back-tracking from a given UC edge, every LO-edge generated during Phase 1 that could be encountered when processing that UC edge is, just in time, generated for further back-tracking. It follows that Phase 1 can, without loss of generality, be viewed as back-tracking along LO-paths in $\mathcal{G}_1$.

(a) $E_u = (U, C{:}{-}u, A)$ is a generated UC edge (b) $E_u = (C, C{:}{-}y, A)$ is an original UC edge

Fig. A.15. Paths relevant to Case a.1 in the proof of Theorem 3.

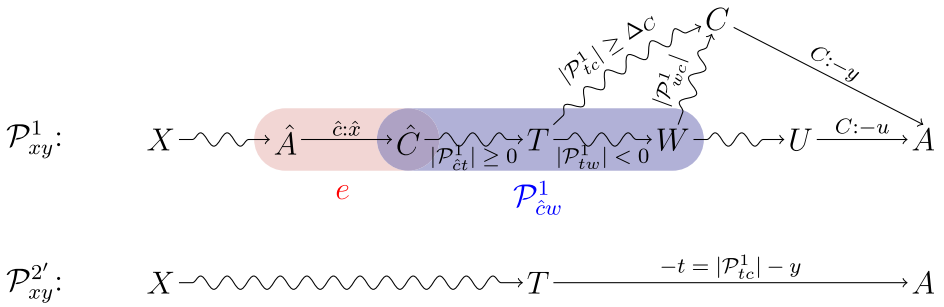


Fig. A.16. Paths relevant to Case a.1 of proof of Theorem 3.

The case where E_u is a *generated* UC edge $(U, C{:}{-}u, A)$ is illustrated in Fig. A.15a, where $\mathcal{P}^1_{tu}$ is the LO-sub-path of $\mathcal{P}^1_{xy}$ from T to U, $\mathcal{P}^1_{uc}$ is any shortest LO-path from U to C in $\mathcal{G}_1$, and the ordinary edge E_t is blue and dashed. (Note that $\mathcal{P}^1_{uc}$ is not a sub-path of $\mathcal{P}^1_{xy}$.) Since $\mathcal{P}^1_{tc}$ is a shortest LO-path from T to C in $\mathcal{G}_1$, it follows that $|\mathcal{P}^1_{tc}| \leq |\mathcal{P}^1_{tu}| + |\mathcal{P}^1_{uc}|$. As a result, $-t = |E_t| = |\mathcal{P}^1_{tc}| - y \leq |\mathcal{P}^1_{tu}| + |\mathcal{P}^1_{uc}| - y = |\mathcal{P}^1_{tu}| + |E_u| = |\mathcal{P}^1_{tu}| - u$. Therefore, letting $\mathcal{P}^{1'}_{xy}$ be the path in $\mathcal{G}_1$ obtained from $\mathcal{P}^1_{xy}$ by replacing the subpath from T to A by the ordinary edge E_t yields $|\mathcal{P}^{1'}_{xy}| \leq |\mathcal{P}^1_{xy}|$.

The (simpler) case where E_u is the *original* UC edge $(C, C{:}{-}y, A)$ (i.e., where $U = C$) is illustrated in Fig. A.15b, where $\mathcal{P}^1_{tc}$ is a shortest path from T to C in $\mathcal{G}_1$, and $\mathcal{P}^{1a}_{tc}$ is the LO-sub-path of $\mathcal{P}^1_{xy}$ from T to C. ($\mathcal{P}^{1a}_{tc}$ and $\mathcal{P}^1_{tc}$ may or may not be the same path.) Since $\mathcal{P}^1_{tc}$ is by definition a shortest LO-path in $\mathcal{G}_1$, it follows that $-t = |E_t| = |\mathcal{P}^1_{tc}| - y \leq |\mathcal{P}^{1a}_{tc}| - y$. Therefore, letting $\mathcal{P}^{1'}_{xy}$ be the path in $\mathcal{G}_1$ obtained from $\mathcal{P}^1_{xy}$ by replacing the subpath from T to A by the ordinary edge E_t again yields $|\mathcal{P}^{1'}_{xy}| \leq |\mathcal{P}^1_{xy}|$.

Next, we must show that the time-point T necessarily belongs to $\mathcal{P}^2_{xy}$. Suppose not. The only way that could happen, as illustrated in the top of Fig. A.16, is if, for some LC edge $e = (\hat{A}, \hat{c}{:}\hat{x}, \hat{C})$, T is an interior time-point in an extension subpath $\mathcal{P}^1_{\hat{c}w}$ in $\mathcal{P}^1_{xy}$, where Phase 2 used e and $\mathcal{P}^1_{\hat{c}w}$ to generate an ordinary bypass edge from $\hat{A}$ to W in $\mathcal{P}^2_{xy}$. In the figure, the LC edge e is highlighted in pink, and the extension sub-path $\mathcal{P}^1_{\hat{c}w}$ is highlighted in purple. In this case, the bypass edge from $\hat{A}$ to W would not only bypass the LC edge, it would also bypass T.

Since the forward propagation of Phase 2 stops when a negative subpath is found, every *proper prefix* of the extension sub-path $\mathcal{P}^1_{\hat{c}w}$, including the subpath $\mathcal{P}^1_{\hat{c}t}$ from $\hat{C}$ to T, must have non-negative length, and every *suffix* of $\mathcal{P}^1_{\hat{c}w}$, including the subpath $\mathcal{P}^1_{tw}$ from T to W, must have negative length.[7] Next, since $\Delta_C \leq |\mathcal{P}^1_{tc}|$, and $\mathcal{P}^1_{tc}$ is a shortest LO-path in $\mathcal{G}_1$, it follows that $\Delta_C \leq |\mathcal{P}^1_{tc}| \leq |\mathcal{P}^1_{tw}| + |\mathcal{P}^1_{wc}| < |\mathcal{P}^1_{wc}|$, since $|\mathcal{P}^1_{tw}| < 0$. But then $\Delta_C < |\mathcal{P}^1_{wc}|$, which contradicts the choice of T. Therefore, T must appear in $\mathcal{P}^2_{xy}$; and the path $\mathcal{P}^{2'}_{xy}$, illustrated at the bottom of Fig. A.16, obtained by replacing the subpath from T to A in $\mathcal{P}^2_{xy}$ by the ordinary edge E_t necessarily belongs to $\mathcal{G}_2$. Furthermore, a similar argument confirms that every LC edge preceding T in $\mathcal{P}^1_{xy}$ that does not belong to $\mathcal{P}^{2'}_{xy}$ must have an extension subpath that *precedes* T in $\mathcal{P}^1_{xy}$ (and $\mathcal{P}^{1'}_{xy}$) and therefore can be used to generate the corresponding ordinary bypass edge in $\mathcal{P}^{2'}_{xy}$. And since Phase 2 is length-preserving, $|\mathcal{P}^{2'}_{xy}| = |\mathcal{P}^{1'}_{xy}| \leq |\mathcal{P}^1_{xy}|$.

Next, let $\mathcal{P}^{3'}_{xy}$ be the path in $\mathcal{G}_3$ obtained by replacing each ordinary edge of $\mathcal{P}^{2'}_{xy}$ by the equivalent ordinary edge-or-path generated by Phase 3. Since Phase 3 is length-preserving, it follows that $|\mathcal{P}^{3'}_{xy}| = |\mathcal{P}^{2'}_{xy}| = |\mathcal{P}^{1'}_{xy}| \leq |\mathcal{P}^1_{xy}|$. Next, every LC or UC edge in $\mathcal{P}^{3'}_{xy}$ also appears in $\mathcal{P}^3_{xy}$; however, $\mathcal{P}^3_{xy}$, unlike $\mathcal{P}^{3'}_{xy}$, also contains the UC edge E_u, as well as any LC edges bypassed by E_t. Therefore, it follows that $\|\mathcal{P}^{3'}_{xy}\|_\omega \leq \|\mathcal{P}^3_{xy}\|_\omega = |\mathcal{P}^\omega_{xy}|$. (Projections preserve the lengths of ordinary edges/paths, but if F is an LC or UC edge, $|F| \leq \|F\|_\omega$, by Lemma 1.)

[7] These kinds of properties of extension sub-paths for LC edges were originally observed and proven by Morris [23].

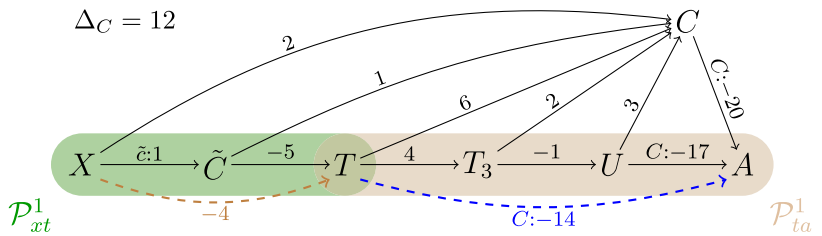


Fig. A.17. An example of Case a.2 of proof of Theorem 3.

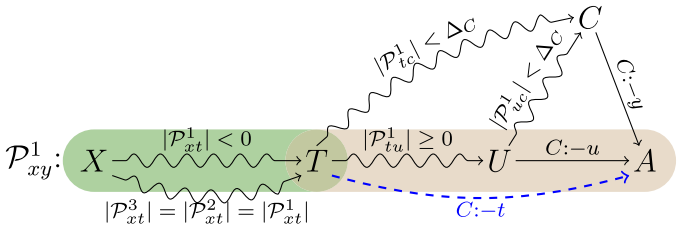


Fig. A.18. General scenario for Case a.2 of proof of Theorem 3.

As a result, the UC edge E_u has effectively been removed from $\mathcal{P}^3_{xy}$, without having increased the length of the projection. (This concludes Case a.1.)

Case a.2: For each T_i, the shortest LO-path $\mathcal{P}^1_{t_ic}$ satisfies $|\mathcal{P}^1_{t_ic}| < \Delta_C$. In this case, let T be the *leftmost* time-point T_i in $\mathcal{P}^1_{xy}$ such that the LO-subpath $\mathcal{P}^1_{tu}$ from T to U is non-negative. (If all such sub-paths are negative, then choose $T = U$.) Fig. A.17 shows a sample scenario for this case, where $\Delta_C = 12$; $\mathcal{P}^1_{xt}$, the LO-subpath of $\mathcal{P}_1$ from X to T is shaded green; and $\mathcal{P}^1_{ta}$, the subpath from T to A, is shaded brown. In the figure, note that the path from T to U has the non-negative length 3, while the paths from $\tilde{C}$ to U, and from $\tilde{X}$ to U, have the negative lengths -1 and -2, respectively. Now, since each T_i is reachable by back-propagation from C, and the length of each $\mathcal{P}^1_{t_ic}$ is less than Δ_C, it follows that Phase 1 must have generated, for each T_i, a UC bypass edge $(T_i, C{:}{-}t_i, A)$ which, by definition, must belong to $\mathcal{G}_1$. In particular, the UC bypass edge $(T, C{:}{-}t, A)$, shown as a blue, dashed edge, must belong to $\mathcal{G}_1$. (Fig. A.18 shows a more general scenario, albeit assuming that $E_u = (U, C{:}{-}u, A)$ is a generated UC edge. The case where E_u is the original UC edge $(C, C{:}{-}y, A)$ is simpler.) In addition, as in Case a.1, T cannot be the interior time-point in the extension sub-path for any LC edge $\hat{e} = (\hat{A}, \hat{c}{:}\hat{x}, \hat{C})$ preceding T in $\mathcal{P}^1_{xy}$ since the prefix from $\hat{C}$ to T would have to be non-negative, implying that the subpath from $\hat{C}$ to U must be non-negative, contradicting the choice of T. As a result, T must belong to $\mathcal{P}^2_{xy}$ and, hence, also to $\mathcal{P}^3_{xy}$.

Therefore, the projected length of $E_t = (T, C{:}{-}t, A)$ satisfies:

$$
\begin{aligned}
\|E_t\|_\omega &= \max\{-t, -\omega_c\} \\
&= \max\{|\mathcal{P}^1_{tc}| - y, -\omega_c\} \\
&\leq \max\{|\mathcal{P}^1_{tu}| + |\mathcal{P}^1_{uc}| - y, -\omega_c\} \\
&= \max\{|\mathcal{P}^1_{tu}| - u, -\omega_c\} \\
&\leq \max\{|\mathcal{P}^1_{tu}| - u, |\mathcal{P}^1_{tu}| - \omega_c\} && \text{since } |\mathcal{P}^1_{tu}| \geq 0 \\
&= |\mathcal{P}^1_{tu}| + \max\{-u, -\omega_c\} \\
&= |\mathcal{P}^1_{tu}| + \|E_u\|_\omega \\
&= |\mathcal{P}^3_{tu}| + \|E_u\|_\omega && \text{Phases 2 and 3 are length preserving} \\
&\leq \|\mathcal{P}^3_{ta}\|_\omega && (*)
\end{aligned}
$$

where, in the last two steps, $\mathcal{P}^3_{tu}$ is the subpath of $\mathcal{P}^3_{xy}$ from T to U, and $\mathcal{P}^3_{ta}$ is the subpath of $\mathcal{P}^3_{xy}$ from T to A. In short, the projected length of E_t is no more than the projected length of the corresponding subpath from T to A in $\mathcal{P}^3_{xy}$.

Next, consider the subpath $\mathcal{P}^1_{xt}$ from X to T. Now, by the choice of T, every suffix of $\mathcal{P}^1_{xt}$ must be negative. Therefore, for each LC edge in $\mathcal{P}^1_{xt}$, if any, Phase 2 generates an ordinary bypass edge, such as the dashed brown edge $(X, -4, T)$ in Fig. A.17. Therefore, let $\mathcal{P}^2_{xt}$ be the *ordinary* path in $\mathcal{G}_2$ obtained from $\mathcal{P}^1_{xt}$ by bypassing each of its LC edges with ordinary edges generated during Phase 2; and let $\mathcal{P}^3_{xt}$ be the equivalent *ordinary vee-path* from X to T guaranteed to exist in $\mathcal{G}_3$ as a result of Phase 3. (Note that $\mathcal{P}^2_{xt}$ and $\mathcal{P}^3_{xt}$ need not be sub-paths of $\mathcal{P}^2_{xy}$ and $\mathcal{P}^3_{xy}$, respectively.) Then, since $\mathcal{P}^3_{xt}$ has only ordinary edges, whereas the subpath $\mathcal{P}^{3^*}_{xt}$ of $\mathcal{P}^3_{xy}$ from X to T may have had some LC edges, Lemma 1 ensures that:

$$\|\mathcal{P}^3_{xt}\|_\omega \leq \|\mathcal{P}^{3^*}_{xt}\|_\omega \ (\dagger)$$

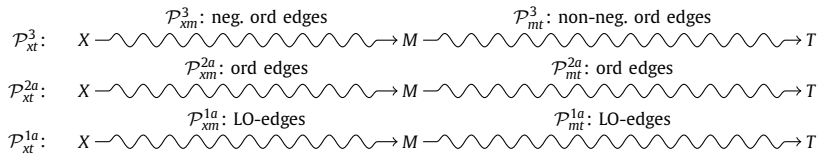


Fig. A.19. Paths relevant to Case a.2.2 of proof of Theorem 3.

Now $\mathcal{P}^3_{xt}$ is a vee-path in $\mathcal{G}_3$. Hence, it consists of zero or more negative ordinary edges, followed by zero or more non-negative ordinary edges.

Case a.2.1: $\mathcal{P}^3_{xt}$ consists solely of negative edges. In this case, let $\mathcal{P}^{3'}_{xy}$ be the path in $\mathcal{G}_3$ that is the concatenation of $\mathcal{P}^3_{xt}$, the UC edge $E_t = (T, C{:}{-}t, A)$, and the suffix $\mathcal{P}^3_{ay}$ of $\mathcal{P}^3_{xy}$ from A onward. Then:

$$\begin{aligned}
\|\mathcal{P}^{3'}_{xy}\|_\omega &= \|\mathcal{P}^3_{xt}\|_\omega + \|E_t\|_\omega + \|\mathcal{P}^3_{ay}\|_\omega \\
&\le \|\mathcal{P}^{3^*}_{xt}\|_\omega + \|\mathcal{P}^3_{ta}\|_\omega + \|\mathcal{P}^3_{ay}\|_\omega && \text{By } (\dagger) \text{ and } (*), \text{ above} \\
&= \|\mathcal{P}^3_{xy}\|_\omega
\end{aligned}$$

In short, $\mathcal{P}^{3'}_{xy}$ has effectively replaced the portion of $\mathcal{P}^3_{xy}$ from X to A by a path consisting of zero or more negative ordinary edges, followed by the UC bypass edge E_t, without increasing the length of its projection.

Case a.2.2: $\mathcal{P}^3_{xt}$ consists of zero or more negative edges followed by one *or more non-negative edges.* In this case, let M be the time-point dividing the negative edges of $\mathcal{P}^3_{xt}$ from the non-negative edges, as illustrated in Fig. A.19, and let $\mathcal{P}^3_{xm}$ and $\mathcal{P}^3_{mt}$ be the subpaths from X to M, and from M to T that comprise all of the negative and non-negative edges of $\mathcal{P}^3_{xt}$, respectively. Given that Phase 3 may replace one ordinary path with a different, but equivalent ord path, it may be that M does not belong to $\mathcal{P}^2_{xt}$. However, each ordinary edge in $\mathcal{P}^3_{xt}$ is equivalent to *some* ordinary path in $\mathcal{G}_2$; hence, as shown in Fig. A.19, there is some path $\mathcal{P}^{2a}_{xt}$ in $\mathcal{G}_2$ that can be obtained by replacing each ordinary edge in $\mathcal{P}^3_{xt}$ by an equivalent ord-edge-or-path in $\mathcal{G}_2$. Under this construction, $\mathcal{P}^{2a}_{xt}$ necessarily contains M. Let $\mathcal{P}^{2a}_{xm}$ and $\mathcal{P}^{2a}_{mt}$ be the sub-paths from X to M, and from M to T, respectively. Similarly, there must be some path $\mathcal{P}^{1a}_{xt}$ in $\mathcal{G}_1$ that can be obtained by replacing each ordinary edge in $\mathcal{P}^{2a}_{xt}$ that was generated during Phase 2 by the corresponding LO-subpath that generated it. Under this construction, $\mathcal{P}^{1a}_{xt}$ necessarily contains the time-point M. Thus, let $\mathcal{P}^{1a}_{xm}$ and $\mathcal{P}^{1a}_{mt}$ be the sub-paths from X to M, and from M to T, respectively. Note that $\mathcal{P}^{1a}_{xt}$ need not be a sub-path of $\mathcal{P}^1_{xy}$.

Now, since $|\mathcal{P}^{1a}_{mt}| = |\mathcal{P}^{2a}_{mt}| = |\mathcal{P}^3_{mt}| \ge 0$, it is possible that, during the Phase 1 processing of the original UC edge $E = (C, C{:}{-}y, A)$, back-propagation from $(T, C{:}{-}t, A)$ along LO-edges in $\mathcal{P}^{1a}_{mt}$ might have encountered a time-point V for which $\mathcal{P}^1_{vc}$, the shortest LO-path from V to C in $\mathcal{G}_1$, satisfied $|\mathcal{P}^1_{vc}| \ge \Delta_C$, leading to the generation of an ordinary edge $(V, -v, A)$ that bypasses both E_t and E. If so, let $\mathcal{P}^{1'}_{xy}$ be the concatenation of: (1) the prefix $\mathcal{P}^{1a}_{xv}$ of $\mathcal{P}^{1a}_{xt}$ from X to V; (2) the ordinary bypass edge $(V, -v, A)$; and (3) the suffix of $\mathcal{P}^1_{xy}$ from A onward. In this way, $\mathcal{P}^{1'}_{xy}$ has a prefix from X to A that contains only LO-edges, as in Case a.1. (A similar analysis as that done in Case a.1 ensures that the projection of the corresponding path $\mathcal{P}^{3'}_{xy}$ will be no greater than $|\mathcal{P}^\omega_{xy}|$.)

Otherwise, all suffixes of $\mathcal{P}^{1a}_{mt}$ must have been explored by Phase 1, with associated shortest LO-paths to C of length less than Δ_C, generating corresponding UC bypass edges, including a UC edge, $E_m = (M, C{:}{-}m, A)$, from M to A that bypasses both $\mathcal{P}^{1a}_{mt}$ and E_t. In this case, let $\mathcal{P}^{1'}_{xy}$ be the concatenation of: (1) the LO-subpath $\mathcal{P}^{1a}_{xm}$ from X to M; (2) the UC bypass edge $E_m = (M, C{:}{-}m, A)$; and (3) the suffix of $\mathcal{P}^1_{xy}$ from A onward. Similarly, let $\mathcal{P}^{2'}_{xy}$ be the concatenation of: (1) the ordinary subpath $\mathcal{P}^{2a}_{xm}$ from X to M, (2) the UC bypass edge E_m, and (3) the suffix of $\mathcal{P}^2_{xy}$ from A onward. Finally, let $\mathcal{P}^{3'}_{xy}$ be the concatenation of: (1) the ordinary subpath $\mathcal{P}^3_{xm}$ from X to M, which consists solely of negative edges; (2) the UC edge E_m; and (3) the suffix of $\mathcal{P}^3_{xy}$ from A onward. Then $\|\mathcal{P}^{3'}_{xy}\|_\omega \le \|\mathcal{P}^3_{xy}\|_\omega$, as follows. First, $\|E_m\|_\omega$ satisfies:

$$\begin{aligned}
\|E_m\|_\omega &= \max\{-m, -\omega_c\} \\
&= \max\{|\mathcal{P}^1_{mc}| - y, -\omega_c\} \\
&\le \max\{|\mathcal{P}^1_{mt}| + |\mathcal{P}^1_{tc}| - y, -\omega_c\} && \text{Shortest path} \\
&= \max\{|\mathcal{P}^{1a}_{mt}| - t, -\omega_c\} \\
&\le \max\{|\mathcal{P}^{1a}_{mt}| - t, |\mathcal{P}^{1a}_{mt}| - \omega_c\} && |\mathcal{P}^{1a}_{mt}| \ge 0 \\
&= |\mathcal{P}^{1a}_{mt}| + \max\{-t, -\omega_c\} \\
&= |\mathcal{P}^{1a}_{mt}| + \|E_t\|_\omega \\
&= |\mathcal{P}^3_{mt}| + \|E_t\|_\omega && \text{Phases 2 and 3 are length preserving} \\
&\le |\mathcal{P}^3_{mt}| + \|\mathcal{P}^3_{ta}\|_\omega && \text{By } (*) \text{ above} \\
&\le \|\mathcal{P}^3_{mt}\|_\omega + \|\mathcal{P}^3_{ta}\|_\omega && \text{By Lemma 1 } (**)
\end{aligned}$$

Thus,

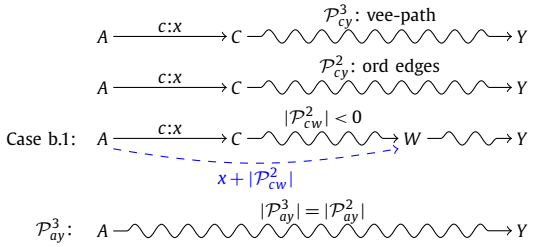


Fig. A.20. Paths in part b of proof of Theorem 3.

$$
\begin{aligned}
\|\mathcal{P}^{3'}_{xy}\|_\omega &= \|\mathcal{P}^3_{xm}\|_\omega + \|E_m\|_\omega + \|\mathcal{P}^3_{ay}\|_\omega \\
&\le \|\mathcal{P}^3_{xm}\|_\omega + (\|\mathcal{P}^3_{mt}\|_\omega + \|\mathcal{P}^3_{ta}\|_\omega) + \|\mathcal{P}^3_{ay}\|_\omega && \text{By } (**), \text{ above} \\
&= \|\mathcal{P}^3_{xt}\|_\omega + \|\mathcal{P}^3_{ta}\|_\omega + \|\mathcal{P}^3_{ay}\|_\omega \\
&= \|\mathcal{P}^{3^*}_{xt}\|_\omega + \|\mathcal{P}^3_{ta}\|_\omega + \|\mathcal{P}^3_{ay}\|_\omega && \text{By } (\dagger), \text{ above} \\
&= \|\mathcal{P}^3_{xy}\|_\omega
\end{aligned}
$$

In short, $\mathcal{P}^{3'}_{xy}$ has effectively replaced the portion of $\mathcal{P}^3_{xy}$ from X to A by a path consisting of zero or more negative ordinary edges, followed by the UC bypass edge E_m, without increasing the length of its projection. This concludes Case a.2.

Recursively analyzing the remaining UC edges in $\mathcal{P}^{3'}_{xy}$. If the leftmost UC edge E_u in $\mathcal{P}^3_{xy}$ is handled by Case a.1, then $\mathcal{P}^{3'}_{xy}$ effectively bypasses E_u, thereby enabling the next leftmost UC edge to be addressed—whether by Case a.1 or a.2. On the other hand, if E_u is handled by Case a.2, then the prefix $\mathcal{P}^3_{xa}$ of $\mathcal{P}^3_{xy}$ from X to A has been replaced in $\mathcal{P}^{3'}_{xy}$ by a sequence of zero or more negative ordinary edges, followed by a UC bypass edge. Hereinafter, such paths may be called nU-subpaths (for "negative, then upper-case"). As a result, the next leftmost UC edge can now be addressed—whether by Case a.1 or a.2—but only considering the portion of $\mathcal{P}^{3'}_{xy}$ from A onward. Continuing in this way, *all* of the UC edges in $\mathcal{P}^3_{xy}$ can eventually be processed, with Cases a.1 and a.2 appearing in any combination, resulting in a modified path $\mathcal{P}^{3u}_{xy}$ which either (1) has no UC edges—which only happens if all UC edges were handled by Case a.1—or (2) has all of its UC edges in a prefix that is the concatenation of one or more nU-subpaths. These two possibilities may be concisely described as $\mathcal{P}^{3u}_{xy}$ having all of its (zero or more) UC edges within a prefix that is the concatenation of zero or more nU-subpaths.

Analyzing the LC edges in $\mathcal{P}^{3u}_{xy}$. Let $\mathcal{P}^{3u}_{xq}$ denote the prefix of $\mathcal{P}^{3u}_{xy}$ described above that contains all of the (zero or more) UC edges from $\mathcal{P}^{3u}_{xy}$ and consists of a sequence of zero or more nU-subpaths, terminating at some time-point Q. Since nU-subpaths only contain ordinary and UC edges, it follows that any LC edges remaining in $\mathcal{P}^{3u}_{xy}$ necessarily belong to the suffix $\mathcal{P}^{3u}_{qy}$ from Q to Y. Furthermore, this suffix is identical to the suffix $\mathcal{P}^3_{qy}$ of $\mathcal{P}^3_{xy}$. Therefore, the following analysis shall begin with $\mathcal{P}^3_{qy}$, and the corresponding suffix $\mathcal{P}^2_{qy}$ of $\mathcal{P}^2_{xy}$.

The analysis of LC edges in $\mathcal{P}^3_{qy}$ is in many ways the mirror image of the analysis of UC edges described earlier. Instead of processing UC edges from left to right, the LC edges are processed from right to left. In addition, the propagation goes *forward* from LC edges; and the roles of negative and non-negative edges are swapped. However, the analysis of LC edges is vastly simpler, mainly because UC edges play no role and Phase 2 only generates ordinary edges, never additional LC edges.

Let $e = (A, c{:}x, C)$ be the rightmost LC edge in $\mathcal{P}^3_{qy}$, and let $\mathcal{P}^3_{cy}$ be the suffix of $\mathcal{P}^3_{qy}$ from C to Y, as illustrated in the top row of Fig. A.20. By the choice of e, all edges in $\mathcal{P}^3_{cy}$ are necessarily ordinary. Furthermore, since $\mathcal{G}_3$ is vee-path-complete, we may assume without loss of generality that $\mathcal{P}^3_{cy}$ is a vee-path. Let $\mathcal{P}^2_{cy}$ be the corresponding path in $\mathcal{G}_2$ obtained by replacing each ordinary edge in $\mathcal{P}^3_{cy}$ by its equivalent ordinary edge-or-path in $\mathcal{G}_2$, as shown in the second row of Fig. A.20.

Case b.1: The first edge of $\mathcal{P}^3_{cy}$ is negative. In this case, $\mathcal{P}^2_{cy}$ necessarily contains a negative-length prefix, as illustrated in the third row of Fig. A.20, where $\mathcal{P}^2_{cw}$ denotes the first negative prefix of $\mathcal{P}^2_{cy}$. By the choice of $\mathcal{P}^2_{cw}$, it follows that all proper prefixes of $\mathcal{P}^2_{cw}$ have non-negative length. As a result, Phase 2 would necessarily generate the ordinary bypass edge $(A, x + |\mathcal{P}^2_{cw}|, W)$, shown as blue and dashed.[8]

Let $\mathcal{P}^2_{ay}$ be the path in $\mathcal{G}_2$ from A to Y obtained by replacing both the LC edge e and the extension sub-path $\mathcal{P}^2_{cw}$ by the ordinary bypass edge from A to W. Note that $\mathcal{P}^2_{ay}$ comprises only ordinary edges. Next, let $\mathcal{P}^3_{ay}$ be the ordinary path in $\mathcal{G}_3$ from A to Y, shown in the last row of Fig. A.20, obtained by applying Phase 3 to the edges in $\mathcal{P}^2_{ay}$. Finally, let $\mathcal{P}^{3'}_{qy}$ be the same as $\mathcal{P}^3_{qy}$, except that the portion from A to Y has been replaced by $\mathcal{P}^3_{ay}$. In this way, the LC edge e has effectively been removed from the path $\mathcal{P}^3_{qy}$. Furthermore, since Phases 2 and 3 are length-preserving and $\mathcal{P}^{3'}_{qy}$ contains one fewer LC edge than $\mathcal{P}^3_{qy}$, Lemma 1 ensures that $\|\mathcal{P}^{3'}_{qy}\|_\omega \le \|\mathcal{P}^3_{qy}\|_\omega$.

[8] Although Phase 2 is applied to LO-paths in $\mathcal{G}_1$, it may equivalently be viewed as applying to LO-paths in $\mathcal{G}_2$. For example, if, when propagating forward from an LC edge e, the extension sub-path $\mathcal{P}^2_{cw}$ contained some ordinary edges in $\mathcal{G}_2$ that were generated by Phase 2 *after* the LC edge e was processed, the forward propagation from e explores all of those same LO-paths, resulting in the same set of bypass edges for e.

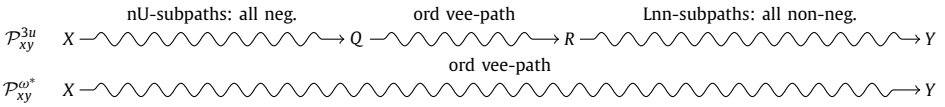


Fig. A.21. Final paths from X to Y in proof of Theorem 3.

Case b.2: The first edge in $\mathcal{P}^3_{cy}$ is non-negative. Since $\mathcal{P}^3_{cy}$ is vee-path, it follows that the path $\mathcal{P}^3_{ay}$ from A to Y consists solely of non-negative edges: in particular, e, followed by non-negative ordinary edges. This kind of path is hereinafter called an Lnn-subpath (for "lower-case followed by non-negative edges"). Therefore, in this case, we can let $\mathcal{P}^{3'}_{qy} = \mathcal{P}^3_{qy}$, and the recursive processing of the next rightmost LC edge can be carried out, but restricting attention to the portion of $\mathcal{P}^{3'}_{qy}$ that *precedes* A.

Recursively analyzing the remaining LC edges in $\mathcal{P}^{3'}_{qy}$. If the LC edge e was bypassed by an ordinary edge as in Case b.1 above, then the next rightmost LC edge in $\mathcal{P}^{3'}_{qy}$ can be recursively processed in the context of the entire path $\mathcal{P}^{3'}_{qy}$. On the other hand, if the LC edge remains in $\mathcal{P}^{3'}_{qy}$ as part of an Lnn-subpath from A to Y, then the next rightmost LC edge can be recursively processed in the context of the portion of $\mathcal{P}^{3'}_{qy}$ that *precedes* A.

After processing all of the LC edges in this way, using whatever combination of Cases b.1 and b.2 are needed, the path from Q to Y will have a suffix from some R to Y that contains all of the remaining LC edges (if any) and is the concatenation of zero or more Lnn-subpaths. Furthermore, since Cases b.1 and b.2 both preserve the length of the overall path, but may remove some LC edges, it follows that the projected length of the overall path from X to Y cannot be longer than the original $\|\mathcal{P}^3_{xy}\|_\omega$.

Concluding the proof.

After processing all of the UC and LC edges from $\mathcal{P}^3_{xy}$, the modified path, shown as $\mathcal{P}^{3u}_{xy}$ in Fig. A.21, will have a prefix from X to Q that is the concatenation of zero or more nU-subpaths that together contain all of the UC edges (if any), and a suffix from R to Y that is the concatenation of zero or more Lnn-subpaths that together contain all of the LC edges (if any). In between, there will be a path from Q to R consisting of zero or more ordinary edges which, given the STN-dispatchability of $\mathcal{G}_3$, can be assumed to be a vee-path. Furthermore, all of the path modifications used in processing/bypassing the UC and LC edges have been shown to result in a projection in $\mathcal{G}_\omega$ that is no longer than the original projection $\|\mathcal{P}^3_{xy}\|_\omega = |\mathcal{P}^\omega_{xy}|$.

Since every edge in an nU-subpath is negative, and every edge in an Lnn-subpath is non-negative, it follows that the projected path, called $\mathcal{P}^{\omega^*}_{xy}$ in Fig. A.21, is a vee-path that is no longer than the original path $\mathcal{P}^\omega_{xy}$. □

References

[1] J. Agrawal, W. Chi, S. Chien, G. Rabideau, D. Gaines, S. Kuhn, Analyzing the effectiveness of rescheduling and Flexible Execution methods to address uncertainty in execution duration for a planetary rover, Robot. Auton. Syst. 140 (2021) 103758, https://doi.org/10.1016/j.robot.2021.103758.

[2] R. Barták, R.A. Morris, K.B. Venable, An introduction to constraint-based temporal reasoning, volume 8. Morgan, https://doi.org/10.2200/S00557ED1V01Y201312AIM026, 2014.

[3] N. Bhargava, C. Muise, T. Vaquero, B. Williams, Delay Controllability: Multi-Agent Coordination under Communication Delay, Technical Report MIT-CSAIL-TR-2018-02, MIT, 2018.

[4] J.L. Bresina, A.K. Jónsson, P.H. Morris, K. Rajan, Activity planning for the Mars exploration rovers, in: Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), 2005, pp. 40–49, https://ntrs.nasa.gov/api/citations/20050157091/downloads/20050157091.pdf.

[5] M. Cairo, L. Hunsberger, R. Rizzi, Faster dynamic controllability checking for simple temporal networks with uncertainty, in: 25th International Symposium on Temporal Representation and Reasoning (TIME-2018), 2018, pp. 8:1–8:16, https://doi.org/10.4230/LIPIcs.TIME.2018.8.

[6] S. Choudhury, J.K. Gupta, M.J. Kochenderfer, D. Sadigh, J. Bohg, Dynamic multi-robot task allocation under uncertainty and temporal constraints, Auton. Robots 46 (2022) 231–247, https://doi.org/10.1007/s10514-021-10022-9.

[7] C. Combi, R. Posenato, Towards temporal controllabilities for workflow schemata, in: 17th Intern. Symp. on Temporal Representation and Reasoning (TIME-2010), 2010, pp. 129–136, https://doi.org/10.1109/TIME.2010.17.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009, http://mitpress.mit.edu/books/introduction-algorithms.

[9] R. Dechter, I. Meiri, J. Pearl, Temporal constraint networks, Artif. Intell. 49 (1991) 61–95, https://doi.org/10.1016/0004-3702(91)90006-6.

[10] J. Eder, M. Franceschetti, J. Köpke, Controllability of business processes with temporal variables, in: 34th ACM/SIGAPP Symposium on Applied Computing, 2019, pp. 40–47, https://doi.org/10.1145/3297280.3297286.

[11] J. Eder, M. Franceschetti, J. Lubas, Dynamic controllability of processes without surprises, Appl. Sci. 12 (2022) 1461, https://doi.org/10.3390/app12031461.

[12] C. Fang, A.J. Wang, B.C. Williams, Chance-constrained static schedules for temporally probabilistic plans, J. Artif. Intell. Res. 75 (2022) 1323–1372, https://doi.org/10.1613/jair.1.13636.

[13] M. Fisher, D.M. Gabbay, L. Vila (Eds.), Handbook of Temporal Reasoning in Artificial Intelligence, 1st ed., Elsevier, 2005.

[14] M. Franceschetti, J. Eder, Computing ranges for temporal parameters of composed web services, in: 21st International Conference on Information Integration and Web-Based Applications & Services, 2019, pp. 537–545, https://doi.org/10.1145/3366030.3366068.

[15] M. Franceschetti, J. Eder, Negotiating temporal commitments in cross-organizational business processes, in: 27th International Symposium on Temporal Representation and Reasoning (TIME-2020), Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, pp. 4:1–4:15, https://doi.org/10.4230/LIPIcs.TIME.2020.4.

[16] M. Gini, Multi-robot allocation of tasks with temporal and ordering constraints, in: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, 2017, pp. 4863–4869, https://doi.org/10.1609/aaai.v31i1.11145.

[17] L. Hunsberger, Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies, in: 16th International Symposium on Temporal Representation and Reasoning (TIME-2009), 2009, pp. 155–162, https://doi.org/10.1109/TIME.2009.25.
[18] L. Hunsberger, R. Posenato, Speeding up the RUL$^-$ dynamic-controllability-checking algorithm for simple temporal networks with uncertainty, in: 36th AAAI Conference on Artificial Intelligence (AAAI-22), AAAI Press, 2022, pp. 9776–9785, https://doi.org/10.1609/aaai.v36i9.21213.
[19] E. Karpas, S.J. Levine, P. Yu, B.C. Williams, Robust execution of plans for human-robot teams, in: 25th Int. Conf. on Automated Planning and Scheduling (ICAPS-15), 2015, pp. 342–346.
[20] A. Lanz, R. Posenato, C. Combi, M. Reichert, Controllability of time-aware processes at run time, in: On the Move to Meaningful Internet Systems (OTM-2013), in: LNCS, vol. 8185, Springer, 2013, pp. 39–56, https://doi.org/10.1007/978-3-642-41030-7_4.
[21] A. Lanz, R. Posenato, C. Combi, M. Reichert, Simple temporal networks with partially shrinkable uncertainty, in: 6th International Conference on Agents and Artificial Intelligence (ICAART 2015), 2015, pp. 370–381, https://doi.org/10.5220/0005200903700381.
[22] A. Lanz, M. Reichert, Enabling time-aware process support with the atapis toolset, in: BPM Demo Sessions 2014, 2014, pp. 41–45.
[23] P. Morris, A structural characterization of temporal dynamic controllability, in: Principles and Practice of Constraint Programming (CP-2006), 2006, pp. 375–389, https://doi.org/10.1007/11889205_28.
[24] P. Morris, Dynamic controllability and dispatchability relationships, in: CPAIOR 2014, in: LNCS, vol. 8451, Springer, 2014, pp. 464–479, https://doi.org/10.1007/978-3-319-07046-9_33.
[25] P. Morris, The mathematics of dispatchability revisited, in: 26th International Conference on Automated Planning and Scheduling (ICAPS-2016), 2016, pp. 244–252.
[26] P. Morris, N. Muscettola, T. Vidal, Dynamic control of plans with temporal uncertainty, in: IJCAI 2001: Proc. of the 17th International Joint Conference on Artificial Intelligence, 2001, pp. 494–499.
[27] P.H. Morris, N. Muscettola, Temporal dynamic controllability revisited, in: 20th National Conference on Artificial Intelligence (AAAI-2005), 2005, pp. 1193–1198, https://www.aaai.org/Papers/AAAI/2005/AAAI05-189.pdf.
[28] N. Muscettola, P. Morris, B. Pell, B. Smith, Issues in temporal reasoning for autonomous control systems, in: Proceedings of the 2nd International Conference on Autonomous Agents - AGENTS '98, 1998, pp. 362–368, https://doi.org/10.1145/280765.280862.
[29] N. Muscettola, P.H. Morris, I. Tsamardinos, Reformulating temporal plans for efficient execution, in: 6th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR-1998), 1998, pp. 444–452.
[30] M. Nilsson, J. Kvarnstrom, P. Doherty, EfficientIDC: a faster incremental dynamic controllability algorithm, in: 24th International Conference on Automated Planning and Scheduling (ICAPS-14), 2014, pp. 199–207.
[31] J. Peng, J. Zhu, L. Zhang, Generalizing STNU to model non-functional constraints for business processes, in: 2022 International Conference on Service Science (ICSS), IEEE, 2022, pp. 104–111, https://doi.org/10.1109/ICSS55994.2022.00024.
[32] R. Posenato, STNU Benchmark version 2020, https://profs.scienze.univr.it/~posenato/software/cstnu/benchmarkWrapper.html, 2020, last access 2022-12-01.
[33] R. Posenato, CSTNU tool: a Java library for checking temporal networks, SoftwareX 17 (2022) 100905, https://doi.org/10.1016/j.softx.2021.100905.
[34] G. Ramalingam, J. Song, L. Joskowicz, R.E. Miller, Solving systems of difference constraints incrementally, Algorithmica 23 (1999) 261–275, https://doi.org/10.1007/PL00009261.
[35] M. Saint-Guillain, T. Stegun Vaquero, J. Agrawal, S. Chien, Robustness computation of dynamic controllability in probabilistic temporal networks with ordinary distributions, in: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, International Joint Conferences on Artificial Intelligence Organization, 2020, pp. 4168–4175, https://doi.org/10.24963/ijcai.2020/576, ISSN: 10450823.
[36] J. Shah, J. Stedl, P. Robertson, B. Williams, A fast incremental algorithm for maintaining dispatchability of partially controllable plans, in: M. Boddy, M. Fox, S. Thiébaux (Eds.), Seventeenth Int. Conf. on Automated Planning and Scheduling (ICAPS-2007), 2007.
[37] J.A. Shah, B.C. Williams, C. Breazeal, Dynamic execution of temporal plans for temporally fluid human robot teaming, in: 2010 AAAI Spring Symposium Series, AAAI Press, 2010, pp. 46–51, https://cdn.aaai.org/ocs/1105/1105-5946-1-PB.pdf.
[38] J. Stedl, B. Williams, A fast incremental dynamic controllability algorithm, in: ICAPS Workshop on Plan Execution: A Reality Check, 2005, pp. 69–75.
[39] I. Tsamardinos, N. Muscettola, P. Morris, Fast transformation of temporal plans for efficient execution, in: 15th National Conf. on Artificial Intelligence (AAAI-1998), 1998, pp. 254–261.