Information and Computation 307 (2025) 105356

Contents lists available at ScienceDirect

Information and Computation

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

Recent algorithmic advances in simple temporal networks with uncertainty: From faster controllability checking to faster execution

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

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

ARTICLE INFO

Article history:
Received 8 April 2025
Received in revised form 2 September 2025
Accepted 12 September 2025
Available online 16 September 2025

Keywords:
Temporal constraint networks
Overconstrained networks
Negative cycles
Dynamic controllability
Dispatchability
Real-time execution
Temporal reasoning
Diamond structures

ABSTRACT

This paper advances the state of the art in the dynamic controllability (DC) and dispatchability of Simple Temporal Networks with Uncertainty (STNUs) through four key contributions.

First, `findSRNC` is an algorithm that identifies semi-reducible negative cycles in non-dynamically controllable STNUs. Running in $O(mn + k^2n + kn\log n)$ time (matching the fastest DC-checking algorithms), it handles repeated edges and uses polynomial space, even when cycles might contain exponentially many edges.

Second, $\texttt{minDisp}^{+}_{\text{ESTNU}}$ is an algorithm that improves dispatchability computation for STNUs from $O(kn^3)$ to $O(n^3 + k^2n\log n)$ time. It outputs dispatchable Extended STNUs (ESTNUs) having minimal numbers of edges, which is crucial for subsequent real-time execution.

Third, the Canonical Form of Nested Diamond Structures in Dispatchable ESTNUs is a rigorous theory that facilitates correctness proofs for dispatchability algorithms. It also helped reveal and correct a flaw in a previously published algorithm.

Fourth, our empirical evaluation using improved open-source implementations demonstrates the practical effectiveness of our algorithms.

These contributions address fundamental computational bottlenecks in temporal planning systems, enabling more efficient reasoning about uncertain timing constraints while providing real-time guarantees required for robotics, scheduling, and automated planning applications.

1. Introduction

Temporal reasoning is a foundational area in Artificial Intelligence (AI), playing a critical role in applications ranging from planning and scheduling to robotics and autonomous systems. Among the various formalisms developed for temporal reasoning, Simple Temporal Networks (STNs) have emerged as a widely used model due to their ability to concisely represent

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

https://doi.org/10.1016/j.ic.2025.105356

and efficiently reason about temporal constraints. However, many real-world scenarios involve inherent uncertainty, such as tasks with durations influenced by external factors or uncontrollable events. To address this challenge, Simple Temporal Networks with Uncertainty (STNUs) extend STNs by incorporating *contingent links* that represent temporal durations that are determined by external processes beyond the system's control. This extension not only increases the expressive power of the model but also introduces new reasoning challenges, such as ensuring controllability in the presence of uncertainty, which are critical for robust and reliable AI systems.

The rest of this section provides relevant background information from the literature on STNs and STNUs, highlighting key properties and the challenges they pose, and addressing recent advances in the field, focusing on the development of efficient algorithms for checking the *dynamic controllability* of STNUs and for computing equivalent dispatchable networks having a minimal number of constraints/edges. These advances have significantly improved the scalability and applicability of STNUs, enabling STNU-based applications to handle larger and more complex temporal reasoning problems efficiently. Interleaved with the overview is a roadmap for the rest of the paper.

1.1. Simple temporal networks

A *Simple Temporal Network* (STN) is a data structure for representing and reasoning about time. STNs are attractive for planning and scheduling applications because they accommodate a variety of common types of constraints (e.g., duration constraints, deadlines, and inter-action constraints) [11,59,15,46,75,28,1,9,45,52,14,51,50,25,4,66].

An STN is *consistent* if there exists a complete set of assignments of values to its timepoints that together satisfy all of its constraints. In other words, an STN is consistent if it has a solution as a constraint satisfaction problem (CSP) [8]. Each STN has a graphical form whose nodes represent timepoints and whose weighted directed edges correspond to the STN's constraints. An STN is consistent if and only if its graph has no negative cycles [21]. The consistency of an STN can be checked in polynomial time (e.g., by the $O(mn)$-time Bellman-Ford algorithm [19]).

A consistent STN can admit different solutions; that is, for each timepoint, there can be a range of possible values that can be assigned depending on the values chosen for the other timepoints. Since assigning a value to a timepoint frequently corresponds to initiating or terminating the execution of an action, assigning values to the timepoints of an STN is often referred to as *executing* the STN. In real scenarios, the ability to execute an STN incrementally as time progresses (i.e., assigning values to one timepoint at a time, in *real time*) is considered to be important because it provides greater *flexibility* in determining the eventual solution. For example, for some timepoints, it may be important to choose the latest possible execution time, while for others, the earliest possible time may be preferred. Determining the updated range of admissible values for a timepoint during real-time execution requires considering the values already assigned to other timepoints and any constraints remaining to be satisfied. For this reason, it is also important to preserve maximal flexibility while requiring minimal real-time computation during execution.

Toward that end, the *dispatchability* of an STN has been defined in terms of performance guarantees with respect to a real-time execution algorithm [72]. The fastest algorithm for converting consistent STNs into an equivalent dispatchable form is the $O(mn + n^2 \log n)$-time algorithm due to Tsamardinos et al. [72], where n is the number of timepoints and m is the number of constraints. In 2016, Morris [55] gave a graphical characterization of STN dispatchability that is useful for proving theorems about dispatchability algorithms. Section 3.1 summarizes some of the definitions and results for STNs that will be important for this paper.

1.2. Simple temporal networks with uncertainty

Although STNs have found wide use in applications over the past 30 years, there have been many extensions of STNs that accommodate additional features that are important in real-world domains. Among these, *Simple Temporal Networks with Uncertainty* (STNUs) have attracted a great deal of theoretical and practical interest (cf. Section 2). STNUs augment STNs to include *contingent links* that can be used to represent actions with uncertain durations (e.g., taxi rides or a visit to the British Museum) [56]. Each STNU has a graphical form that augments an STN graph to include additional edges, called *Lower-Case* (LC) and *Upper-Case* (UC) edges, that represent bounds on the uncontrollable action durations. Section 3.2 summarizes the definitions and results for STNUs that will be important for this paper.

For many applications, the most important property of an STNU is called *dynamic controllability* (DC). An STNU is DC if there exists a dynamic strategy for executing its controllable timepoints that guarantees that all relevant constraints will be satisfied no matter how the uncertain durations turn out—within their specified bounds [56,30]. There are many polynomial-time algorithms, called DC-checking algorithms, for determining whether any given STNU is DC. The fastest is the $O(mn + k^2 n + kn \log n)$-time `RUL`$^-$ algorithm due to Cairo et al. [12], where n is the number of timepoints; m, the number of constraints; and k, the number of contingent links. Hunsberger and Posenato subsequently presented a modification of `RUL`$^-$, called `RUL2021`, that has the same worst-case complexity, but is an order of magnitude faster in practice [36]. Section 4 summarizes the operation of the `RUL2021` algorithm.

The characteristic feature of a non-DC STNU is that it must contain a *semi-reducible negative* (SRN) cycle [53]. In general, any path from X to Y in an STNU graph is semi-reducible if it entails a path of the same length from X to Y that contains no LC edges. Such entailments can be discovered by generating new edges using constraint-propagation (equivalently, edge-generation) rules. Although finding negative cycles in STNs reduces to finding *simple* negative cycles (i.e., no repeat nodes),

finding SRN cycles in STNUs is more complex, given that even *indivisible* SRN cycles in a non-DC STNU can have repeat edges and, in the worst case, an *exponential* number of such edges [34]. (An SRN cycle is indivisible if each proper sub-cycle is non-negative or non-semi-reducible.)

When given a *non*-DC STNU, DC-checking algorithms simply report that the network is not DC; they do not produce an SRN cycle [53,54,12,36]. For applications, it is important to identify SRN cycles so that they can be repaired (e.g., by accepting the cost of weakening constraints or tightening uncertain durations). Existing algorithms for finding SRN cycles in non-DC STNUs [76,77,74,2,6] are based on older, less efficient DC-checking algorithms; and the issue of repeated edges has been ignored or given scant attention.

⇒ Section 5 presents the first novel contribution of this paper: a new, faster algorithm, called `findSRNC`, for computing SRN cycles in non-DC STNUs while also rigorously addressing the compact representation of SRN cycles having a large number of repeated edges.[1] The `findSRNC` algorithm modifies the `RUL2021` DC-checking algorithm to accumulate path information without impacting its time complexity. The additional space required to compactly store path information, while avoiding redundant storage of repeated edges, is $O(mk + k^2n)$.

As with STNs, it is important to preserve maximal flexibility while requiring minimal computation during the execution of an STNU. However, typical *DC-checking algorithms* presented in the literature only confirm the *existence* of a dynamic execution strategy; they do not output one [53,54,12,36]. Now fully specifying a dynamic execution strategy typically requires exponential space and, therefore, exponential time, since it is necessary to store all possible reactions to all possible contingent executions. As a result, instead of storing all possible reactions, *incremental execution strategies* have been developed that can *react* to observations of contingent executions as they occur, propagating information across the rest of the network [56,31,32,35,13].

The challenge for incremental execution strategies is to preserve maximum flexibility during execution while requiring minimal real-time computation. Toward that end, and as described in Section 6.2, the notion of dispatchability has also been defined for STNUs [54]. However, unlike for STNs, the dispatchability of an STNU had not been specified in terms of performance guarantees with respect to a particular real-time execution algorithm, but instead in terms of the dispatchability of its STN *projections*. (A projection of an STNU is the STN that results from assigning a fixed duration to each contingent link.) Given the graphical characterization of STN dispatchability, this definition of STNU dispatchability is attractive from a theoretical perspective. However, it had only been argued informally that dispatchability for an STNU, defined in this way, would provide the desired performance guarantees in the context of real-time execution [54].

To correct this disconnect, recent work [41] proceeded as follows. First, it defined a real-time execution algorithm for STNUs, called `RTE`*, that preserves maximal flexibility while requiring minimal computation. Next, it proved that an STNU is dispatchable (according to the graphical definition) if and only if every run of the `RTE`* algorithm necessarily satisfies all of the STNU's ordinary constraints, no matter how the uncertain durations turn out. In this way, that work filled an important gap in the foundations of STNU dispatchability. Section 6 gives detailed background on the progression from dispatchability for STNs to dispatchability for STNUs and Extended STNUs (ESTNUs).

Despite these advances, until recently, algorithms for converting an STNU into an equivalent dispatchable form [54,37] did not provide any guarantees about the size of the resulting network. Since the number of edges in a dispatchable network directly impacts the amount of computation required during real-time execution, it is desirable to find an equivalent dispatchable ESTNU having a minimal number of edges, what we call the *MinDispESTNU* problem. The first algorithm for solving the MinDispESTNU problem is the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm [39], which has a worst-case time-complexity of $O(kn^3)$. Section 7 of this paper describes the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm.

The remaining novel contributions of this paper are as follows:

⇒ Appendix A presents a novel theoretical analysis of the *canonical form of nested diamond structures* which it then uses to rigorously confirm the correctness of $\texttt{minDisp}_{\text{ESTNU}}$. The analysis in Appendix A also reveals an error in a later algorithm, called $\texttt{fastMinDisp}_{\text{ESTNU}}$ [40], that invalidates its faster performance.[2]

⇒ Section 7 presents a new version of the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm, called $\texttt{minDisp}^{+}_{\text{ESTNU}}$, that corrects the error in $\texttt{fastMinDisp}_{\text{ESTNU}}$ by replacing a substantial algorithmic component, while also more efficiently dealing with nested diamond structures, resulting in a worst-case time-complexity of $O(n^3 + k^2 n \log n)$, which is the best so far for solving the MinDispESTNU problem.

⇒ Section 8 presents an empirical evaluation of all of the algorithms introduced in the paper.

Section 9 gives concluding remarks.

[1] The `findSRNC` algorithm was originally presented in a conference paper [38]. The presentation here expands on that work.

[2] The theoretical analysis of the canonical form of nested diamond structures was originally presented in a technical report [44].

⇒ Pseudocode for all relevant algorithms is presented in detail since the authors believe in the importance of providing comprehensive algorithmic specifications to ensure unambiguous reproducibility and to facilitate accurate complexity analysis.

2. Applications of simple temporal networks with uncertainty

Simple Temporal Networks with Uncertainty (STNUs) are applied across domains where managing temporal constraints under uncertainty is critical. Below we highlight the most relevant applications.

Planning and scheduling under uncertainty. STNUs are widely used and extended in planning and scheduling, especially when task durations or execution times are affected by uncontrollable external factors. For example, in multicore computing, making efficient use of on-chip resources requires scheduling that is robust to delays; STNUs can be used to schedule tasks effectively despite such delays [49]. In multi-agent systems, each agent may maintain its own plan while control over activity durations is shared; in this case, each plan can be modeled as a separate STNU [69]. Finally, STNUs often serve as a base model for more sophisticated planning frameworks, such as probabilistic networks, where uncertainty is represented using probability distributions [24,29].

Business process modeling. In business process modeling, STNUs are used to represent workflows with uncertain task durations and to ensure that key temporal constraints hold despite variability [17]. For instance, STNUs have been applied to flexible business processes where contingent durations depend on external conditions, such as resource availability or environmental factors [62,23,27,22,65,26,18,48,47]. These models help process designers analyze and verify the dynamic controllability of workflows and support reliable execution under uncertainty.

Robotics and autonomous systems. In robotics, STNUs model and manage task executions that involve uncontrollable durations or external events. Robotic systems operating in dynamic environments often rely on STNUs to reason about action timing and to adapt schedules in real time [67,5,20], enabling robots to achieve goals while coping with unpredictable delays or variations in execution.

Temporal reasoning in artificial intelligence. STNUs are used in broader AI contexts, including temporal reasoning and knowledge representation. They serve as a core model for representing temporal relationships and ensuring consistency when durations are uncertain [3]. For example, STNUs have been used to approximate the probability of successful execution in probabilistic temporal networks, aiding decision-making in complex AI systems [42].

Healthcare and patient scheduling. STNUs have also been applied in healthcare, notably for patient scheduling and treatment planning. They are used to model and optimize temporal workflows in clinical environments, where task durations and patient conditions introduce significant uncertainty [48,47]. Such applications aim to ensure timely delivery of care while accommodating variability in healthcare processes.

3. From STN consistency to STNU dynamic controllability

For expository purposes, this section begins with a basic scenario that will be used as a running example throughout the rest of the paper.

Repair Scenario. A technician has been assigned to do a mechanical repair job at a client's house. The job must be done (start to finish) between 10:00 a.m. and 12:00 noon. The technician's shift starts at 9:00 a.m. The drive from the technician's warehouse to the client's house will take between 20 and 40 minutes, depending on traffic. The technician is required to start the job no more than 10 minutes after arriving at the client's house. The job itself will take between 30 and 90 minutes, depending on the complexity of repair. The technician must complete the job no more than 135 minutes after leaving the warehouse. This scenario includes two actions with uncertain-but-bounded durations: the drive from the warehouse to the client's house, and the repair job itself. The technician only controls when to leave the warehouse and when to start the job.

Since Simple Temporal Networks (STNs) cannot accommodate actions with uncertain durations, the following restricted scenario is presented for comparison purposes.

Restricted Repair Scenario. *Same as above, except that the technician controls the duration of the drive, as well as the duration of the job.*

3.1. Simple temporal networks

This section gives an overview of Simple Temporal Networks (STNs) from which all other kinds of temporal networks are derived.

Table 1
Sample temporal networks with differences highlighted in blue. (For interpretation of the colors in the table(s), the reader is referred to the web version of this article.)

$\mathcal{T} = \{Z, D, E, R, J\}$

$$\mathcal{C} = \left\{ \begin{array}{lll} Z - D \le -540, & E - R \le 0, & R - E \le 10, \\ Z - R \le -600, & J - Z \le 720, & J - D \le 135, \\ D - E \le -20, & E - D \le 40, & \\ R - J \le -30, & J - R \le 90 & \end{array} \right\}$$

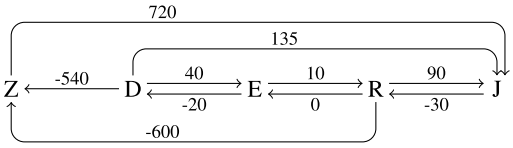
(a) A Simple Temporal Network for the Restricted Repair Scenario

$\mathcal{T} = \{Z, D, E, R, J\}$

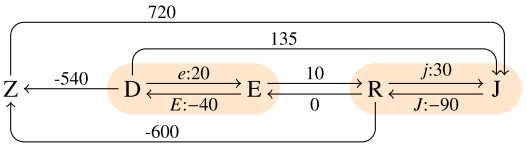
$$\mathcal{C} = \left\{ \begin{array}{lll} Z - D \le -540, & E - R \le 0, & R - E \le 10, \\ Z - R \le -600, & J - Z \le 720, & J - D \le 135 \end{array} \right\}$$

$\mathcal{L} = \{(D, 20, 40, E), (R, 30, 90, J)\}$

(b) A Simple Temporal Network with Uncertainty for the Repair Scenario



(a) The graph for the STN from Table 1a



(b) The graph for the STNU from Table 1b

Fig. 1. Graphs for the temporal networks from Table 1 with contingent links shaded orange. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

A *Simple Temporal Network* (STN) is a pair, $(\mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ is a set of real-valued variables called *timepoints* (TPs) and $\mathcal{C}$ is a set of binary difference constraints, called *ordinary constraints,* each of the form $Y - X \le \delta$, where $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$ [21]. The timepoints typically represent starting or ending times of actions; the constraints can represent deadlines, release times, and duration or inter-action constraints. With no loss of generality, it is convenient to assume that each STN has a special timepoint Z whose value is fixed at *zero* (or some other convenient timestamp) and is constrained to occur at or before every other timepoint.[3] Typically, we let $n = |\mathcal{T}|$ and $m = |\mathcal{C}|$.

Table 1a shows an STN representing the Restricted Repair Scenario in which the technician controls the durations of the drive and repair job. The STN's timepoints are Z (midnight, fixed at 0), D (the start of the drive), E (the end of the drive), R (the start of the repair), and J (the end of the job). All times are expressed in minutes. For example, the constraint, $Z - D \le -540$ (i.e., $D \ge 540$) ensures that the technician's drive starts at or after 9:00 a.m. (i.e., the beginning of the technician's shift). The constraints, $E - R \le 0$ and $R - E \le 10$ (i.e., $R - E \in [0, 10]$) represent that the repair must begin no more than 10 minutes after the technician arrives at the client's house. $Z - R \le -600$ (i.e., $R \ge 600$) and $J - Z \le 720$ (i.e., $J \le 720$) together ensure that the job is done between 10:00 a.m. and 12:00 noon. Finally, $J - D \le 135$ ensures that the job is finished no more than 135 minutes after the technician leaves the warehouse. The constraints, $D - E \le -20$ and $E - D \le 40$ together ensure that the drive duration satisfies $E - D \in [20, 40]$. Similarly, $R - J \le -30$ and $J - R \le 90$ ensure that the job duration satisfies $J - R \in [30, 90]$.

The STN in Table 1a is consistent. For example, one of its solutions is: $Z = 0$ (midnight), $D = 570$ (9:30 a.m.), $E = R = 600$ (10:00 a.m.), and $J = 660$ (11:00 a.m.). Notice that in this solution to the Restricted Repair Scenario, the technician *chose* the duration of the drive to be 30 minutes, and the duration of the repair to be 60 minutes, which of course limits the practicality of the restricted scenario.

Each STN has a corresponding graph, $(\mathcal{T}, \mathcal{E})$, where the timepoints in $\mathcal{T}$ serve as nodes and each constraint $Y - X \le \delta$ in $\mathcal{C}$ corresponds to a labeled directed edge $X \xrightarrow{\delta} Y$ in $\mathcal{E}$, called an *ordinary edge.* For convenience, such edges may be notated as (X, δ, Y). Fig. 1a shows the graph for the STN from Table 1a.[4] In general, an STN is consistent if and only if its graph has no negative cycles [21]. It is not hard to check that all of the cycles in the STN graph in Fig. 1a have non-negative length.

Several well-known algorithms play important roles in the work presented in this paper, including: Bellman-Ford, Dijkstra, and Johnson's algorithm [19]. The $O(mn)$-time Bellman-Ford algorithm can be used to generate a solution for a consistent STN. Such a solution can then be used to transform the edge-weights in an STN graph to all non-negative values. (When used in this way, a solution f is typically called a *potential function.*) That, in turn, enables Dijkstra's $O(m + n \log n)$-time Single-Source Shortest-Paths (SSSP) algorithm to be used. Johnson's algorithm uses Bellman-Ford to generate a potential function and then applies Dijkstra n times, once for each timepoint as source node, to do an All-Pairs Shortest-Paths (APSP) computation in $O(mn + n^2 \log n)$ time. The corresponding n-by-n matrix of APSP distances is called the *distance matrix* for

[3] It is not hard to show that in any consistent STN there is at least one timepoint that can play the role of Z.

[4] An alternative representation of STN graphs labels edges with *intervals.* For example, an edge from X to Y labeled by the interval $[a, b]$ would represent the constraint, $Y - X \in [a, b]$. In our paper, this constraint would correspond to the pair of edges, (X, b, Y) and $(Y, -a, X)$, representing the pair of constraints, $Y - X \le b$ and $X - Y \le -a$. One drawback of labeling edges with intervals is that each constraint has two equivalent representations as edges. For example, the constraint $E - D \in [-20, 40]$ can be represented by an edge from D to E labeled by $[-20, 40]$ or an edge from E to D labeled by $[-40, 20]$. Similarly, a one-sided constraint such as $J - Z \le 720$ can be represented by an edge from Z to J labeled by $(-\infty, 720]$ or an edge from J to Z labeled by $[-720, \infty)$.

the STN. Its values represent the strongest constraints that *every* solution to the STN must satisfy. The distance matrix can also be used to identify an inconsistent network.

3.2. Simple temporal networks with uncertainty

A *Simple Temporal Network with Uncertainty* (STNU) augments an STN to include *contingent links* that can be used to represent actions with uncertain durations [56]. Formally, 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 contingent link has the form (A, x, y, C) where $A, C \in \mathcal{T}$ and $0 < x < y < \infty$ [56]. A is called the *activation* TP and C the *contingent* TP; and we let $\Delta_C = y - x$. Contingent links represent *uncontrollable possibilities* in the sense that an executor typically controls only the execution of A, but not C. The executor only *observes* the execution of C in real-time, knowing only that C will be executed such that $C - A \in [x, y]$. Considering the above Repair Scenario, the drive made by the technician can be represented by the contingent link $(D, 20, 40, E)$, where D is when the technician starts to drive, E is when the drive ends, and $E - D \in [20, 40]$ is the uncertain duration, learned only when the drive is finished. We let $k = |\mathcal{L}|$; and notate the set of contingent timepoints as $\mathcal{T}_c$; and the non-contingent (i.e., executable) timepoints as $\mathcal{T}_x = \mathcal{T} \backslash \mathcal{T}_c$. Table 1b shows an STNU representing the Repair Scenario with two contingent links: 1) $(D, 20, 40, E)$ to represent the drive action and 2) $(R, 30, 90, J)$ to represent the repair action. In this STNU, $n = 5, m = 6$ and $k = 2$.

Each STNU $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ has a corresponding graph, $(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u)$, where: $(\mathcal{T}, \mathcal{E}_o)$ is the graph for the STN $(\mathcal{T}, \mathcal{C})$; $\mathcal{E}_l$ is a set of *lower-case* (LC) edges; and $\mathcal{E}_u$ is a set of *upper-case* (UC) edges. The LC and UC edges correspond to the contingent links in $\mathcal{L}$, as follows. For each contingent link $(A, x, y, C) \in \mathcal{L}$, there is an LC edge $A \xrightarrow{c:x} C$ in $\mathcal{E}_l$ and a UC edge $C \xrightarrow{C:-y} A$ in $\mathcal{E}_u$, respectively representing the *uncontrollable possibilities* that the duration $C - A$ might take on its lower bound x or its upper bound y. For convenience, such LC and UC edges may be notated as $(A, c{:}x, C)$ and $(C, C{:}-y, A)$, respectively. Using this notation, $\mathcal{E}_o = \{(X, \delta, Y) \mid (Y - X) \leq \delta \in \mathcal{C}\}$; $\mathcal{E}_\ell = \{(A, c{:}x, C) \mid (A, x, y, C) \in \mathcal{L}\}$; and $\mathcal{E}_u = \{(C, C{:}-y, A) \mid (A, x, y, C) \in \mathcal{L}\}$. Fig. 1b shows the graph for the STNU from Table 1b. For convenience, we frequently blur the distinction between an STNU and its graph, and between edges and constraints.

An STNU is *dynamically controllable* (DC) if there exists a dynamic strategy for executing its *non-contingent* timepoints such that all of the constraints in $\mathcal{C}$ will necessarily be satisfied no matter how the contingent durations turn out—within their specified bounds [56,30]. A strategy is dynamic in that it can react in real time to observations of contingent executions, but its execution decisions cannot depend on advance knowledge of contingent durations.[5] As is common in the literature, this paper assumes that strategies can react instantaneously to observations [53]. The STNU in Table 1b is DC. One of its valid execution strategies can be glossed as: *"Execute* Z *at* 0 *(midnight) and D at* 570 *(9:30 AM). If E is observed to execute at some time* $t < 35$*, then execute R at* $t + 10$*; otherwise, execute R at* $t + 5$*."* General-purpose algorithms for determining whether any given STNU is DC are called *DC-checking algorithms.* Morris [54] presented the first $O(n^3)$-time DC-checking algorithm for STNUs. Cairo et al. [12] gave a $O(mn + k^2 n + kn \log n)$-time algorithm, called `RUL⁻`, that is faster on sparse networks. It is the DC-checking algorithm with the best worst-case time complexity. However, `RUL2021`, which is a modification of `RUL⁻`, has been shown to be an order-of-magnitude faster on a variety of STNU benchmarks, although having the same theoretical complexity [36]. Section 4 details the `RUL2021` DC-checking algorithm, which plays an important role in the remainder of the paper.

The LO-graph *and the* OU-graph. Certain subsets of labeled edges play important roles in several of the algorithms discussed in this paper. *LO-edges* are either LC or ordinary edges, while *OU-edges* are either ordinary or UC edges. Similarly, *LO-paths* comprise LO-edges, while *OU-paths* comprise OU-edges. Finally, the *LO-graph*, notated as $\mathcal{G}_{\ell o}$, is the subgraph comprising all of the LO-edges, while the *OU-graph*, notated as $\mathcal{G}_{ou}$, is the subgraph comprising the OU-edges. Either of these subgraphs can be viewed as an STN by ignoring the alphabetic labels on its edges. In this way, the Bellman-Ford algorithm can be used to generate a potential function for it, thereby enabling the use of Dijkstra's algorithm (e.g., to update the potential function in response to the insertion of new edges).

4. The `RUL2021` DC-checking algorithm

Because our new `findSRNC` algorithm, presented in Section 5, is closely based on the `RUL2021` DC-checking algorithm [36], this section provides a detailed summary of `RUL2021`.

Like all DC-checking algorithms, the `RUL2021` algorithm operates on the STNU graph, using edge-generation rules to generate new edges representing constraints that must be satisfied by any dynamic execution strategy. Unlike other algorithms that propagate forward along OU-edges aiming to "reduce away" LC edges [53], or propagate backward along

[5] Other forms of controllability have also been studied. An STNU is *weakly controllable* if for each possible combination of contingent durations, there exists a complete assignment of values to non-contingent timepoints that satisfies all of the network's constraints. Crucially, the assignment of values, fixed at the beginning of the execution, can depend on advance knowledge of all contingent durations [73,7,3,68]. In contrast, *strong controllability* requires the existence of a *single* fixed assignment of values to non-contingent timepoints that is chosen before execution begins, but without knowing which combination of contingent durations will eventually be revealed, and yet is guaranteed to satisfy all of the network's constraints [73,57,16,7,3].

Table 2
The edge-generation rules for the RUL2021 algorithm.

Rule	Graphical representation	Applicability Conditions
R	$P \xrightarrow{v} Q \overset{w}{\rightsquigarrow} C_i$ $v+w$	$Q \in \mathcal{T}_X$, $w < \Delta_{C_i}$, $C_i \in \mathcal{T}_C$
L	$A_j \xrightarrow{c_j{:}x_j} C_j \overset{w}{\rightsquigarrow} C_i$ x_j+w	$C_j \not\equiv C_i$, $w < \Delta_{C_i}$, $C_i \in \mathcal{T}_C$
U_{lp}	$P \overset{v}{\rightsquigarrow} C_i \xrightarrow{C_i{:}-y_i} A_i$ $v-y_i$	$(A_i, x_i, y_i, C_i) \in \mathcal{L}$, $v \geq \Delta_{C_i}$

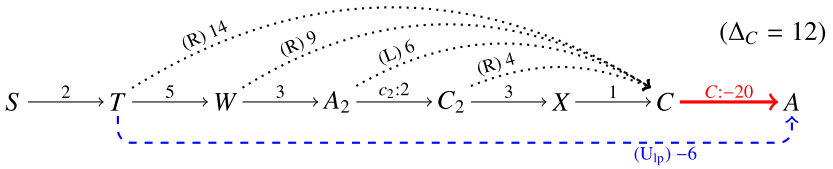


Fig. 2. RUL2021 generating a (blue and dashed) bypass edge for a (red) UC edge.

non-negative LO-edges aiming to "reduce away" negative OU-edges [54], the RUL2021 algorithm, like its predecessor RUL$^-$ [12], propagates backward along LO-edges aiming to "reduce away" UC edges.[6] However, RUL2021 differs from RUL$^-$ in the following ways:

- it restricts its edge generation to *length-preserving* rules;
- it inserts dramatically fewer edges into the STNU graph;
- it is implemented recursively, which facilitates efficiently restarting its processing of a UC edge after one or more interruptions; and
- it sometimes needs to do a limited form of forward propagation in response to certain, rarely encountered structures.

Taken together, these modifications improve the speed of DC checking, while inserting many fewer edges into the STNU graph.

Table 2 shows the edge-generation rules used by RUL2021. The R and L rules (for *Relax* and *Lower Case*, respectively) are used to back-propagate distance information (i.e., path-lengths) in the LO-graph. The wavy arrows represent paths in the LO-graph that have already been explored (i.e., LO-paths whose lengths have already been computed). In the R rule, back-propagation proceeds along the ordinary edge (P, v, Q) to generate the distance information represented by the dotted edge $(P, v+w, C_i)$. In the L rule, back-propagation proceeds along the LC edge $(A_j, c_j{:}x_j, C_j)$ to generate the distance information represented by the dotted edge (A_j, x_j+w, C_i). Unlike the RUL$^-$ algorithm, RUL2021 uses the L and R rules only to accumulate distance information (i.e., path lengths); the dotted edges are *not* inserted into the STNU graph. In contrast, RUL2021 *does* insert the edges generated by the U_{lp} rule: ordinary edges that effectively *bypass* (i.e., reduce away) UC edges. For example, in the table, the wavy path (P, v, C_i) represents distance information previously generated by the R and L rules. This "edge" combines with the UC edge $(C_i, C_i{:}-y_i, A_i)$ to generate the (blue and dashed) bypass edge $(P, v-y_i, A_i)$.

Fig. 2 shows how RUL2021 processes a (red) UC edge associated with a contingent link $(A, 8, 20, C)$, for which $\Delta_C = 20-8 = 12$. First, it uses the R and L rules to back-propagate from C along LO-edges, collecting distance information indicated by the dotted arrows. (The rules used to generate the dotted "edges" are in parentheses.) Back-propagation continues as long as the distance stays less than Δ_C. (See the condition $w < \Delta_i$ for the R and L rules in Table 2.) Since the path from T to C has length $14 \geq \Delta_C$, back-propagation using R and L stops. Next, the U_{lp} rule is applied to $(T, 14, C)$ and $(C, C{:}-20, A)$ to generate the (dashed) bypass edge $(T, -6, A)$. This bypass edge represents a constraint, namely $A - T \leq -6$ (equivalently, $T \geq A+6$), that must be satisfied during execution to guard against the possibility that the contingent duration might take on its maximum value.

[6] "Reducing away" a labeled edge E means generating the constraints entailed by E in combination with other edges so that, afterward, E can be ignored. For example, if all LC edges can be reduced away, then attention can be restricted to the complementary subgraph of (the original and newly generated) OU-edges. Alternatively, if all negative OU-edges can be reduced away, then attention can be restricted to the complementary subgraph of (the original and newly generated) non-negative LO-edges. And, as yet another alternative, if all UC edges can be reduced away, then attention can be restricted to the complementary subgraph of (the original and newly generated) LO-edges. Different approaches focus on reducing away different types of labeled edges, while checking for negative cycles in the complementary subgraph. The RUL$^-$ and RUL2021 algorithms happen to have the best worst-case time complexity.

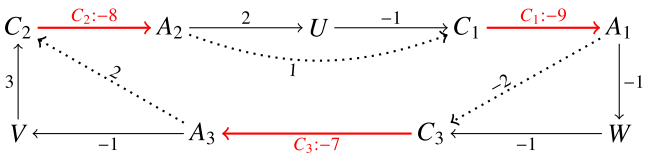


Fig. 3. A cycle of interruptions detected by RUL2021.

Fig. 4. Two different CC loops associated with a contingent link $(A, 1, 9, C)$.

The (dashed, blue) bypass edges generated in this way are the only edges that RUL2021 *may* end up inserting into the STNU graph. However, because there can be many paths emanating from C back to T in the LO-graph, it is important to ensure that only *shortest* distance information is accumulated. For this reason, back-propagation using the R and L rules is governed by a Dijkstra-like process, using a priority queue and a potential function to re-weight the edges in the LO-graph to non-negative values.

The potential function for the LO-graph is initialized by a one-time call to the Bellman-Ford algorithm [19]. After all of the shortest bypass edges for a given UC edge have been computed, they are inserted into the LO-graph, which typically requires updating the potential function. Since all of the new edges terminate at A, this updating can be carried out by a separate Dijkstra-like back-propagation from A using a priority queue and the pre-existing potential function. If all of the UC edges can be successfully processed in this way, then RUL2021 declares the STNU to be DC. However, three kinds of events can signal that the STNU is not DC:

1. **Detecting negative cycles in the LO-graph while attempting to initialize or update the potential function.** A negative cycle in the LO-graph implies that the STNU cannot be DC—because each edge in the LO-graph must be satisfied in the situation where every contingent duration takes on its minimum value. The presence of a negative cycle in the LO-graph can be detected in two ways: (1) by the initial call to Bellman-Ford; or (2) when updating the potential function in response to the addition of new bypass edges.
2. **Cycle of interruptions.** When processing a UC edge $\mathbf{E}_1$, back-propagation from its contingent timepoint C_1 might bump into a different UC edge $\mathbf{E}_2$. If so, RUL2021 *interrupts* its processing of $\mathbf{E}_1$ to process $\mathbf{E}_2$. After finishing with $\mathbf{E}_2$, back-propagation from $\mathbf{E}_1$ continues. However, should a *cycle* of such interruptions occur, for example, as illustrated in Fig. 3, then the network cannot be DC. In the figure, the UC edges are colored red; distances along LO-paths computed by back-propagation using the L and R rules are indicated by dotted arrows; and the relevant contingent links are $(A_1, 1, 9, C_1)$, $(A_2, 2, 8, C_2)$ and $(A_3, 3, 7, C_3)$. Now back-propagation from C_1 should continue as long as the dotted distances are less than $\Delta_{C_1} = 9 - 1 = 8$, but it is interrupted by the UC edge $(C_2, C_2{:}{-}8, A_2)$. Similarly, back-propagation from C_2 should continue as long as the dotted distances are less than $\Delta_{C_2} = 8 - 2 = 6$, but it is interrupted by the UC edge $(C_3, C_3{:}{-}7, A_3)$. Finally, back-propagation from C_3 should continue as long as the dotted distances are less than $\Delta_{C_3} = 7 - 3 = 4$, but it is interrupted by the first UC edge, thereby forming a cycle of interruptions. At this point, RUL2021 signals that the STNU is not DC. This is justified since each dotted distance being less than the corresponding Δ_{C_i} value ensures that the cycle length is negative; and a negative cycle in the OU-graph implies that the STNU cannot be DC because every edge in the OU-graph must be satisfied in the situation where every contingent link takes on its maximum value.
3. **CC loops.** Back-propagation from a UC edge $(C, C{:}{-}y, A)$ can also be blocked if an LO-path from C back to C of length less than Δ_C is encountered. Such a path is called a *CC loop* [36]. A CC loop does not necessarily imply that the STNU is not DC; but it sometimes does. Fig. 4 illustrates two scenarios in which back-propagation from C reveals a CC loop of length $2 < 8 = \Delta_C$. However, the lefthand STNU is not DC, while the righthand one is. The key difference, according to Morris' analysis of semi-reducible paths [53], is that the lefthand graph contains a *negative* LO-path emanating (in the *forward* direction) from C to X that can be used to generate the (dashed, green) bypass edge $(A, -1, X)$, thereby creating a negative cycle in the OU-graph from A to X to C to A, whereas the righthand graph has no such path.

Algorithm 1: The `RUL2021` algorithm.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, an STNU graph with $n = |\mathcal{T}|$ and $k = |\mathcal{E}_u| = |\mathcal{E}_\ell|$
Output: $\top$, if $\mathcal{G}$ is DC; $\bot$, otherwise
Side Effect: Inserts new edges into $\mathcal{G}$

```
1 global := new global struct                                   // global has these fields: pf, status
2 global.pf := BellmanFord(G_lo)                                // pf is a solution for the LO-graph, used as a potential function
3 if global.pf == ⊥ then return ⊥
4 global.status := [notYet, ..., notYet]                        // status is a k-vector giving the processing status of each UC edge
5 foreach E = (C, C:−y, A) ∈ E_u do                              // Try processing each UC edge
6     if RUL2021_RulBackProp(G, E, global) == ⊥ then return ⊥   // If unable to reduce away E, fail
7 return ⊤
```

Algorithm 2: The `RUL2021_RulBackProp` function.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; $\mathbf{E} = (C, C{:}{-}y, A) \in \mathcal{E}_u$, a UC edge; `global`, an instance of a *global* data structure
Output: $\top$, if and only if $\mathbf{E}$ can be successfully processed
Side Effect: Back-propagates from $\mathbf{E}$ along shortest paths in LO-graph, inserting bypass edges into $\mathcal{G}$, and updating `global`

```
1  st := global.status                                          // st is a k-vector containing the processing status for each UC-edge
2  if st[E] == started then return ⊥                            // Cycle of interruptions detected; STNU cannot be DC
3  if st[E] == done then return ⊤                               // E has already been processed (all bypass edges already generated); skip
4  global.status[E] := started                                  // Set processing status of E to started
5  Δ_C := y − x                                                 // Contingent link associated with E is (A, x, y, C)
6  f := global.pf                                               // f is a potential function for the LO-graph
7  loc :=new loc struct          // loc maintains information about the processing of E; its fields are: ccLoop, dist, unstartedUCs
8  loc.ccLoop := ⊥                                              // No CC loop found yet
9  loc.dist := [∞, ..., ∞]       // loc.dist is an n-vector that maintains distance info obtained by back-propagating from C
10 Q := a new priority queue (initialized below)                // key(X) = f(X) + δ_xc = adjusted distance from X to C
11 foreach (X, δ_xc, C) ∈ E_o do Q.insert(X, f(X) + δ_xc)       // Initialize Q to include each X connected to C by an ordinary edge
12 continue? := ⊤
13 while continue? do
14     if !(RUL2021_TryBackProp(G, C, Q, global, loc)) then return ⊥   // Back-propagation processing of E failed
15     if loc.unstartedUCs ≠ ∅ then                             // Back-propagation from E was interrupted by some unstarted UC edge(s)
16         foreach (E_X, X) ∈ loc.unstartedUCs do               // Process each interrupting UC edge; X is the activation TP for E_X
17             if !(RUL2021_RulBackProp(G, E_X, global)) then return ⊥   // Processing an interrupting edge failed
           /* Initialize Q for next iteration; processing E will resume from activation TPs of interrupting edges */
18         Q.clear()
19         foreach (E_X, X) ∈ loc.unstartedUCs do Q.insert(X, loc.dist[X] + global.pf[X])
20     else continue? := ⊥                                      // If got here, no interrupting UC edges; back-propagation from E completed
   /* Check if there was a CC loop with an LO-path that could be used to reduce away the LC edge (A, c:x, C) */
21 if loc.ccLoop and RUL2021_FwdPropNDC(G, C, Δ_C, loc.dist, global.pf) then return ⊥
   /* Insert ordinary edges that bypass E by applying the U_lp rule to all XC "edges" */
22 addedEdges? := ⊥
23 foreach X ∈ T\{C} do                                         // Check each X to see if need to insert bypass edge from X to A
24     δ_xc := loc.dist[X]                                      // Fetch distance from X to C found during back-propataion along LO-edges
25     if Δ_C ≤ δ_xc < ∞ then                                   // δ_xc < ∞ implies that X seen during back-propagation
26         G.insertOrdEdge(X, δ_xc − y, A)                      // δ_xc ≥ Δ_C implies that the U_lp rule can be applied
27         addedEdges? := ⊤
   /* If inserted any bypass edges, then need to update the potential function */
28 if addedEdges? then global.pf := RUL2021_UpdatePotFn(G, A, global.pf)
29 if global.pf == ⊥ then return ⊥
30 st[E] := done
31 return ⊤                                                    // Processing of E successfully completed
```

Pseudocode for the `RUL2021` algorithm is given as Algorithms 1–7. The names of the constituent algorithms have been modified to improve readability and to distinguish them from related algorithms in Section 5.[7] Algorithm 1 (`RUL2021`) initializes a global data structure that contains, `pf`, a potential function for the LO-graph that is initialized by a call to

[7] The translation from new names to their original names [36] is given by: (1) `RUL2021` ⇒ `RUL2021`, (2) `RUL2021_RulBackProp` ⇒ `RULbp`, (3) `RUL2021_TryBackProp` ⇒ `phase1`, (4) `RUL2021_FwdPropNDC` ⇒ `fwdPropNDC`, (5) `RUL2021_UpdatePotFn` ⇒ `UpdPF`, (6) `RUL2021_UpdateVal` ⇒ `UpdVal` (7) `RUL2021_ApplyRL` ⇒ `apRL`.

Bellman-Ford, and `status`, a vector that tracks the processing status of each UC edge. The algorithm then iterates through the UC edges, calling Algorithm 2 (`RUL2021_RulBackProp`) on each one.

`RUL2021_RulBackProp` processes a given UC edge $\mathbf{E} = (C, C{:}{-}y, A)$ as follows. First (at Line 2), if the processing of $\mathbf{E}$ has already been started (by a preceding call) but not yet completed, that entails a cycle of interrupting UC edges, so the algorithm immediately returns $\bot$. Second (at Line 3), if $\mathbf{E}$ has already been successfully processed, the algorithm immediately returns $\top$. Next (at Lines 7-9), it initializes an instance of a *local* data structure, called `loc`, that is used to maintain information that is accumulated across possibly many attempts to back-propagate from $\mathbf{E}$. The `ccLoop` field is a flag that can be set when back-propagation encounters a CC loop; `dist` is a vector that maintains shortest-path information for LO-paths terminating at C; and `unstartedUCs` (initialized by `RUL2021_TryBackProp`, Algorithm 3) accumulates any unstarted UC edges that interrupt back-propagation from $\mathbf{E}$.

Lines 10-11 initialize a priority queue, $\mathcal{Q}$, that is used to guide the Dijkstra-like back-propagation from C in the LO-graph. For each X in the queue, the key is the re-weighted distance, $f(X) + \delta_{xc}$, where f is the potential function for the LO-graph drawn from the *global* data structure (at Line 6) and δ_{xc} is the shortest distance found so far in the LO-graph from X to C.[8] Initially, the queue contains those X that are connected by a single ordinary edge to C.

Each iteration of the `while` loop (Lines 13-20) begins (at Line 14) by calling the `RUL2021_TryBackProp` algorithm (Algorithm 3, discussed below) to carry out the back-propagation from C in the LO-graph using the queue $\mathcal{Q}$. If that back-propagation fails, then `RUL2021_RulBackProp` immediately returns $\bot$. Otherwise, it checks (at Line 15) whether the back-propagation from C was interrupted by any unstarted UC edge(s). If so (at Lines 16-17), it recursively processes each interrupting UC edge and, if successful, prepares the queue (at Lines 18-19) for the next iteration of the `while` loop. A key feature is that the queue is re-initialized so that back-propagation will re-start not from scratch, but from where it was interrupted (i.e., from the activation timepoints of the now-successfully-processed interrupting UC edges). The activation timepoints of the interrupting UC edges are the targets of all as-yet-unexplored paths in the LO-graph. If some iteration of the `while` loop eventually completes (i.e., with no interruptions by unstarted UC edges), then the `while` loop is exited (Line 20).

After the `while` loop, the algorithm checks (at Line 21) whether any CC loops were detected during back-propagation. If so, it calls `RUL2021_FwdPropNDC` algorithm (Algorithm 4, discussed below) to see whether the LC edge (A, x, y, C) can be reduced away by an LO-path. If so, `RUL2021_RulBackProp` immediately returns $\bot$. Otherwise, Lines 22-27 check each timepoint X that was encountered during back-propagation (i.e., each X for which $\delta_{xc} < \infty$) to see whether the U_{lp} rule can be used to generate a bypass edge from X to A, which can only happen if $\delta_{xc} \geq \Delta_C$. If any bypass edges are actually inserted, then (at Line 28) the potential function for the LO-graph is updated by the `RUL2021_UpdatePotFn` algorithm (Algorithm 5, discussed below) and, assuming no negative cycles in the LO-graph are found, the processing status of $\mathbf{E}$ is set to `done`.

As previously mentioned, it is the `RUL2021_TryBackProp` algorithm (Algorithm 3) that actually does the back-propagation along paths in the LO-graph using the data and queue initialized by `RUL2021_RulBackProp`. Each iteration of its `while` loop (Lines 4-18) pops a timepoint X off the queue and uses its key to update the value of `loc`.dist[X] (Lines 5-7). Then (at Line 8), if X happens to be an activation timepoint, $\mathbf{E}_X$ is set to the corresponding UC edge. Next, four cases are considered, only the last of which actually involves back-propagation. Case 1 (Line 10) is where propagation has cycled back to C (i.e., a CC loop has been found). In response, the algorithm simply sets the `ccLoop` flag. Cases 2 and 3 (Lines 11-13) are where back-propagation has been interrupted by an encounter with another UC edge, $\mathbf{E}_X$. In Case 2, the processing of $\mathbf{E}_X$ has not yet been started, so this interrupting UC edge is added to the `loc.unstartedUCs` accumulator. In Case 3, $\mathbf{E}_X$ has already been started, but not yet completed, which signals a cycle of interruptions, implying that the STNU cannot be DC. Finally, Case 4 (Lines 14-18) is where back-propagation actually occurs. The `RUL2021_ApplyRL` algorithm (Algorithm 7) returns a list of pairs specifying candidate applications of the R and L rules to LO-edges incoming to X (e.g., an ordinary edge (W, δ_{wx}, X) or an LC edge $(W, x{:}\delta_{wx}, X)$). Case 4 then uses this information to update the relevant distances stored in the queue. Finally, after `RUL2021_TryBackProp` exhausts the queue, back-propagation is complete (Line 19).

The `RUL2021_FwdPropNDC` algorithm (Algorithm 4) is called whenever back-propagation from a UC edge $(C, C{:}{-}y, A)$ encounters a CC loop (i.e., a cycle in the LO-graph from C back to C where the distance from C back to each participating timepoint W satisfies `dist`$[W] < \Delta_C$). `RUL2021_FwdPropNDC` uses *forward* propagation from C along LO-paths, visiting only those W for which `dist`$[W] < \Delta_C$, aiming to generate an ordinary edge that *bypasses* the corresponding *lower-case* edge $(A, c{:}x, C)$. In particular, if it finds a *negative-length* LO-path $\mathcal{P}$ from C to W, where `dist`$[W] < \Delta_C$, then, following Morris' analysis [53], $\mathcal{P}$ can be used to generate an ordinary edge $(A, x + |\mathcal{P}|, W)$. But then, still following Morris, that edge, together with the LO-path from C to X, and the UC edge $(C, C{:}{-}y, A)$, forms a semi-reducible cycle whose length satisfies:

$$\begin{aligned} (x + |\mathcal{P}|) + \texttt{dist}[W] - y &< x + \texttt{dist}[W] - y && \text{(since } |\mathcal{P}| < 0\text{)} \\ &< x + \Delta_C - y && \text{(since } \texttt{dist}[W] < \Delta_C\text{)} \\ &= 0 && \text{(since } \Delta_C = y - x\text{)} \end{aligned}$$

[8] Typically, the re-weighted distance would be $f(X) + \delta_{xc} - f(C) \geq 0$. Here, the constant term, $-f(C)$, is dropped from each key, which has no meaningful impact on the priority queue.

Algorithm 3: The `RUL2021_TryBackProp` function.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; $C \in \mathcal{T}_c$, a contingent TP; $\mathcal{Q}$, a priority queue; `global`, a *global* struct; `loc`, a *local* struct
Output: $\bot$, iff back-propagation from C reveals $\mathcal{G}$ to be not DC
Side Effect: Propagates backward from C along shortest LO-paths, updating contents of `loc` and `global`, but no change to $\mathcal{G}$

```
1  f := global.pf                                   // f is a potential function, a solution to the LO-graph
2  st := global.status                              // status is a k-vector giving the processing status of each UC edge
3  loc.unstartedUCs := {}                           // Accumulates unstarted UC edges encountered during back-propagation
4  while ¬Q.empty() do
5      (X, key_X) := Q.extractMinNode()             // key_X = distance from X to C, adjusted using potential function
6      δ_xc := key_X − f(X)                         // δ_xc = actual distance from X to C in the LO-graph
7      loc.dist[X] := δ_xc                          // Record the shortest distance from X to C in the LO-graph
8      E_X := G.UCEdgeFromATP(X)                    // If X is an ATP, then E_X is the corresponding UC-edge; otherwise E_X = ⊥
9      if δ_xc < Δ_C then                           // Applicability condition for the R and L rules
10         if X ≡ C then loc.ccLoop := ⊤            // Case 1: Found CC loop of length δ_xc < Δ_C; no further back-propagation
11         else if E_X and st[E_X] == notYet then   // Case 2: E_X is an interrupting unstarted UC-edge
12             loc.unstartedUCs.add((E_X, X))       // Record interruption; no further back-propagation
13         else if E_X and st[E_X] == started then return ⊥   // Case 3: Cycle of interrupting UC edges: not DC
14         else                                     // Case 4: Do back-propagation from X along LO-edges
15             foreach (W, δ_wc) ∈ RUL2021_ApplyRL(G, X, Δ_C, δ_xc) do
16                 newKey := δ_wc + f(W)
17                 if δ_wc < loc.dist[W] and (W ∉ Q or newKey < Q.key(W)) then
18                     Q.insertOrDecreaseKey(W, newKey)   // Update the key for W in the queue to newKey
19 return ⊤                                         // Back-propagation successfully completed
```

Algorithm 4: The `RUL2021_FwdPropNDC` function.

Input: $\mathcal{G}$, an STNU graph; $C \in \mathcal{T}_C$; $\Delta_C = y - x$; `dist`, a vector of XC distances; f, a potential function
Output: $\top$, iff *forward* propagation from C along LO-edges finds path that can be used to bypass the LC edge $(A, c{:}x, C)$

```
1  Q := new priority queue                          // For each W in queue, key(W) = (distance from C forward to W in the LO-graph) − f(W)
2  Q.insert(C, −f(C))                               // Queue initially contains only C
3  while ¬Q.empty() do
4      (W, key_w) := Q.extractMinNode()
5      δ_cw := key_w + f(W)                         // δ_cw = actual distance from C forward to W in the LO-graph
6      if dist[W] < Δ_C then                        // Only explore W if the distance from C backward to W was less than Δ_C
7          if δ_cw < 0 then return ⊤                // Path from C forward to W can bypass (i.e., reduce away) the LC-edge
8          foreach (W, δ_wv, V) ∈ E_ℓ ∪ E_o do      // Else continue forward prop from W along LO-edges, ignoring any LC labels
9              Q.insertOrDecreaseKey(V, δ_cw + δ_wv − f(V))   // Check if key for V needs to be updated to δ_cw + δ_wv − f(V)
10 return ⊥                                         // Was unable to bypass (i.e., reduce away) the LC-edge (A, c:x, C)
```

Hence, that semi-reducible cycle is an SRN cycle. Therefore, `RUL2021_FwdPropNDC` returns $\top$ (i.e., it was able to generate an SRN cycle), which will immediately cause the `RUL2021` algorithm to terminate.

Algorithm 5: The `RUL2021_UpdatePotFn` function.

Input: $\mathcal{G}$, an STNU graph; A, an activation timepoint; f, a potential function for $\mathcal{G}_{\ell o}$, excluding edges ending at A
Output: A potential function f' for all of $\mathcal{G}_{\ell o}$, including edges terminating at A; or $\bot$ if $\mathcal{G}_{\ell o}$ is inconsistent

```
1  f' := copy_vector(f)
2  Q := new empty priority queue                    // For each X ∈ Q, key(X) = Δ_f(X) = f(X) − f'(X)
3  Q.insert(A, 0)                                   // Initial queue for back-propagation from A along LO-edges
4  while ¬Q.empty() do
5      (V, keyV) := Q.extractMinNode()
6      foreach (U, δ, V) ∈ E_o do                   // Back-propagate along ordinary edges ending at V
7          if RUL2021_UpdateVal((U, δ, V), f, f', Q) == ⊥ then return ⊥   // Found a negative cycle in the LO-graph
8      if V ∈ T_C then                              // If V is contingent, back-propagate along the corresponding LC edge, (A_V, v:x_V, V)
9          (A_V, x_V, y_V, V) := contingent link for V
10         if RUL2021_UpdateVal((A_V, x_V, V), f, f', Q) == ⊥ then return ⊥   // Found a negative cycle in the LO-graph
11 return f'
```

Algorithm 6: `RUL2021_UpdateVal` function.

Input: (U, δ, V), an edge; f and f', original and updated potential functions; $\mathcal{Q}$, a priority queue
Output: $\top$ if f' can be updated to satisfy (U, δ, V) without detecting a negative cycle in the LO-graph; $\bot$, otherwise
Side Effect: Modifies $\mathcal{Q}$ and f'

1 **if** $f'(U) < f'(V) - \delta$ **then**
2 $f'(U) := f'(V) - \delta$ *// Increase potential at U to satisfy the edge (U, δ, V)*
3 **if** $\mathcal{Q}.state(U) == alreadyPopped$ **then return** $\bot$ *// Back-prop cycling back to A implies a negative cycle in $\mathcal{G}_{\ell o}$*
4 $\mathcal{Q}.insertOrDecreaseKey(U, f(U) - f'(U))$
5 **return** $\top$

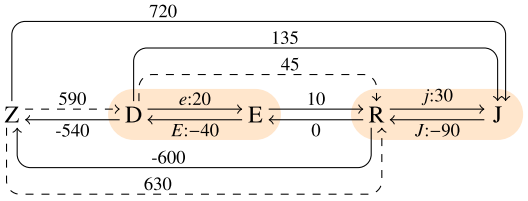


Fig. 5. Repair-Scenario STNU with (dashed) constraints inserted by the `RUL2021` DC-checking algorithm.

If `RUL2021_FwdPropNDC` fails to show that the STNU is *not* DC then, as previously seen, `RUL2021_RulBackProp` (Algorithm 2, Lines 22-27) inserts all bypass edges arising from the processing of the UC edge **E** and, if any are inserted, updates the potential function for the LO-graph, stored in `global.pf` (Line 28). Equivalently to the `RUL`$^-$ algorithm, this updating is handled by the `RUL2021_UpdatePotFn` and `RUL2021_UpdateVal` functions (Algorithms 5 and 6). An important feature is that new bypass edges for a given UC edge $(C, C{:}{-}y, A)$ all terminate at the activation timepoint A. This allows the `RUL2021_UpdatePotFn` function to use the pre-existing potential function, called f, to guide a Dijkstra-like traversal of paths emanating backward from A, where the weight of each edge (U, δ, V) is adjusted to the value $\delta^*_{uv} = f(U) + \delta - f(V)$, as in Johnson's algorithm, except that the adjusted weights may be negative for the new edges terminating at A, since f is not guaranteed to satisfy them. Nonetheless, Dijkstra's algorithm still works for a single-sink back-propagation where the only negative edges are the ones pointing at the sink, A. In addition, when computing the updated values of the potential function, the algorithm utilizes the invariant that the amount of the update at some node U satisfies $\Delta_f(U) = f(U) - f'(U) = d^*(U, A)$, where $d^*(U, A)$ is the shortest distance from U to A in the adjusted-weights graph. In particular, if (U, δ, V) is on a shortest path from U to A, then $\Delta_f(U) = f(U) - f'(U) = f(U) - (f'(V) - \delta)$ (cf. Line 2 in `RUL2021_UpdateVal`). But this equals $(f(U) + \delta - f(V)) + (f(V) - f'(V))$ which, by the inductive hypothesis, equals $\delta^*_{uv} + d^*(V, A) = d^*(U, A)$, since (U, δ, V) is on a shortest path from U to A.

Algorithm 7: The `RUL2021_ApplyRL` function.

Input: $\mathcal{G}$, an STNU graph; $V \in \mathcal{T}$; $\Delta_C = y - x$; and δ_{vc}, the distance from V to C in the LO-graph
Output: A list of timepoint/distance pairs, (W, δ_{wc}), each representing the result of applying the R or L rule to an LO-edge WV and (V, δ_{vc}, C), where $\delta_{wc} = |WV| + \delta_{vc}$ is the distance from W to C via V

1 $pairs := \{\}$
2 **if** $\delta_{vc} < \Delta_C$ **then** *// Application condition for R and L rules*
3 **if** $V \in \mathcal{T}_C$ **then** $pairs.add((A_V, x_V + \delta_{vc}))$ *// If V is contingent, apply the L rule to $(A_V, v{:}x_V, V)$ and (V, δ_{vc}, C)*
4 **else foreach** $(W, \delta_{wv}, V) \in \mathcal{E}_o$ **do** $pairs.add((W, \delta_{wv} + \delta_{vc}))$ *// Else apply the R rule to each (W, δ_{wv}, V) and (V, δ_{vc}, C)*
5 **return** *pairs*

Finally, Case 4 of `RUL2021_TryBackProp` (Algorithm 3, Lines 14-18) calls `RUL2021_ApplyRL` (Algorithm 7) to carry out the back-propagation from X. (To avoid having the expression x_X appear in the pseudocode, the input X is called V in `RUL2021_ApplyRL`.) Given a timepoint V, `RUL2021_ApplyRL` iterates over all incoming LO-edges to V. For each incoming LO-edge, it applies the R or L rule to generate new distance information, collected in pairs of the form, (W, δ_{wc}), where δ_{wc} is the newly discovered distance from W to C via V.

STNU repair scenario checked by `RUL2021`. Fig. 5 shows the STNU representing the Repair Scenario after applying the `RUL2021` algorithm to it. The dashed edges indicate the ordinary constraints inserted into the network by `RUL2021`. Note that even though the STNU is now dynamically controllable, it does not contain enough explicit information to ensure that executing it incrementally will necessarily satisfy all of the original constraints. For example, after executing Z at 0, it seems that the timepoint D could be executed at any time between 540 and 590, whereas its safe execution range is actually $[570, 590]$, as will be shown in Section 6.

5. The `findSRNC` (find semi-reducible negative cycle) algorithm

Given the importance of the DC property (i.e., that it guarantees the existence of a successful dynamic execution strategy), it is important for applications to be able to repair non-DC STNUs, whether by weakening some of their ordinary constraints (e.g., by extending a deadline) or reducing their uncertainty by tightening some of their contingent links (e.g., by paying for a faster, more reliable train service). Since a necessary and sufficient feature of a non-DC STNU is a semi-reducible negative cycle, a straightforward way to restore the DC property is to iteratively identify an SRN cycle, modify some of its constraints or contingent links, and then re-check the DC property. This section concentrates on the first part of this approach: identifying SRN cycles in non-DC STNUs.

Although finding negative cycles in Simple Temporal Networks (STNs) reduces to finding *simple* negative cycles (i.e., no repeat nodes), finding SRN cycles in STNUs is more complex. It has been shown that even *indivisible* SRN cycles in a non-DC STNU can have repeat edges and, in the worst case, an *exponential* number of such edges [34]. In view of this, when given a *non*-DC STNU, most DC-checking algorithms simply report that the network is not DC; they do not produce an SRN cycle [53,54,12,36]. Furthermore, existing algorithms for finding SRN cycles in non-DC STNUs [76,77,74,2,6] are based on older, less efficient DC-checking algorithms; and the issue of repeated edges in SRN cycles has been ignored or given scant attention.

This section introduces our new `findSRNC` algorithm, which modifies the `RUL2021` DC-checking algorithm to efficiently accumulate path information. When given a non-DC STNU, `findSRNC` outputs a *compact* representation of an SRN cycle, which is especially important for SRN cycles having a large number of repeated edges [38]. The time complexity of `findSRNC` is the same as that of `RUL2021`. The additional space required to compactly store path information, while avoiding redundant storage of repeated edges, is $O(mk + k^2 n)$. To contrast `findSRNC` and `RUL2021`, the pseudocode for `findSRNC` presented in this section (in Algorithms 8–15) has the same general structure as `RUL2021`, with modifications highlighted in green.

Recall that there are three different ways that `RUL2021` might detect that the input network is not DC: (1) by failing to initialize or update the potential function for the LO-graph; (2) by encountering a cycle of interruptions of the processing of UC edges; or (3) by encountering a CC loop that contains a subpath that can be used to reduce away (i.e., bypass) the relevant LC edge. The path information collected by the `findSRNC` algorithm ensures that a compact representation for an SRN cycle can be generated in response to any of these possibilities.

`findSRNC` and its helpers use three new data structures to keep track of relevant path information obtained when back-propagating from a UC edge $\mathbf{E} = (C, C{:}{-}y, A)$: (1) `loc.path` is an n-vector that for each timepoint X records the shortest path from X to A seen so far during back-propagation; (2) `global.intBy` is a k-vector that records when a UC edge $\mathbf{E}' = (C', C'{:}{-}y', A')$ interrupts the processing of $\mathbf{E}$, storing both $\mathbf{E}'$ and the explored path from A' to A; and (3) `global.edgeAnnotation` is a hash-table that, for each bypass edge (X, δ, A), records the path used to generate it. A key feature of the path representation is that bypass edges, which can be repeated numerous times in a fully expanded SRN cycle, are not expanded. Since at most kn bypass edges can be generated, this leads to a compact representation.

Algorithm 8: The `findSRNC` algorithm.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, an STNU graph
Output: (*negCycle*, *edgeAnn*), where *negCycle* is an SRN cycle and *edgeAnn* is a hash table of path annotations for its edges; or $(\emptyset, \emptyset)$ if the STNU is DC
Side Effect: Inserts new edges into $\mathcal{G}$

```
1  global := new global data structure                          // New fields: intBy, edgeAnnotation
2  global.pf := BellmanFord(G_lo)
3  if global.pf == ⊥ then return BFCT(G_lo)                     // If LO-graph inconsistent, use BFCT alg. to compute simple negative cycle
4  global.edgeAnnotation := new empty hash table                // Used in Algorithms 9 and 11; hash-key = (X, A), value = path from X to A
5  global.intBy := [⊥, ..., ⊥]                                  // intBy is a k-vector for recording interruptions; used in Algorithms 9-11
6  global.status := [notYet, ..., notYet]
7  foreach E = (C, C:−y, A) ∈ E_u do
8      negCycle := RulBackProp(G, E, global)                    // If recRULbackprop finds an SRN cycle, return it
9      if negCycle ≠ ∅ then return (negCycle, global.edgeAnnotation)   // along with the edgeAnnotation hash table
10 return (∅, ∅);
```

Pseudocode for `findSRNC` is in Algorithm 8. When given a non-DC STNU, it outputs a compact representation of an SRN cycle in the form (*negCycle*, *edgeAnn*), where *negCycle* is a negative cycle of edges in the LO- or OU-graph, depending on how the cycle arose; and *edgeAnn* is a hash table of (*key*, *value*) pairs, where each *key* identifies a bypass edge generated by the algorithm, and *value* is (the possibly compact version of) the path used to generate that edge.

Like `RUL2021`, `findSRNC` starts by calling the Bellman-Ford algorithm to create an initial potential function for the LO-graph. If Bellman-Ford fails, then there must be a *simple* negative cycle in the LO-graph. In that case, treating the LO-graph as an STN, `findSRNC` calls the $O(mn)$-time BFCT algorithm [70] to compute and return a simple negative cycle (Line 3). A simple negative cycle in the LO-graph is a trivial case of an SRN cycle for an STNU.

If Bellman-Ford succeeds, `findSRNC` initializes new fields in the `global` data structure. First, `edgeAnnotation` is set to a new hash table that will record the paths from which any bypass edges are derived. Second, it sets the `intBy` field to a vector of k entries, each initially $\bot$. This vector will store information about when the processing of one UC edge is interrupted by another, which will be used by Algorithms 9-11, discussed below. Finally, like `RUL2021`, `findSRNC`, at Lines 7-9, iterates through the UC edges, calling (in this case) `RulBackProp`(Algorithm 9). If any such call fails, `RulBackProp`will return a negative cycle which `findSRNC` then pairs with the `edgeAnnotation` hash table (Line 9); otherwise, it returns $(\bot, \bot)$ indicating that the input STNU was DC after all (Line 10).

Algorithm 9: The `RulBackProp`algorithm.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; $\mathbf{E} = (C, C{:}{-}y, A) \in \mathcal{E}_u$, a UC edge; `global`, an instance of a *global* data structure
Output: *negCycle*, an SRN cycle; or $\emptyset$ if $\mathbf{E}$ successfully processed
Side Effect: Back-propagates from $\mathbf{E}$ along shortest paths in LO-graph, inserting bypass edges into $\mathcal{G}$, and updating `global`

```
1  st := global.status                                   // st is a k-vector containing the processing status for each UC-edge
2  if st[E] == started then return AccNegCycle (global.intBy,E)   // Cycle of interruptions detected; return SRN cycle
3  if st[E] == done then return ∅                        // E has already been successfully processed
4  global.status[E] := started                           // Prepare to start processing the UC edge E
5  Δ_C := y − x                                          // Contingent link associated with E is (A, x, y, C)
6  f := global.pf                                        // f is a potential function for the LO-graph
7  loc := new local struct                               // Includes new path field
8  loc.ccLoop := ⊥                                       // No CC loop found yet
9  loc.dist := [∞, ..., ∞]                               // distance from each TP to C
10 loc.path := [⟨⟩, ..., ⟨⟩]                             // For each timepoint X, loc.path[X] contains the discovered path from X to A (via E)
11 Q := a new priority queue (initialized below)         // For each X, key(X) = h(X) + δ_xc, adjusted distance from X to C
12 foreach (X, δ_xc, C) ∈ E_o do                         // Initialize Q to include each X connected to C by an ordinary edge
13     Q.insert(X, f(X) + δ_xc)
14     loc.path[X] := (X, δ_xc, C) + (C, C:−y, A)        // loc.path[X] records the two-edge path from X to C to A
15 while continue? do                                    // Start or resume processing of UC edge E
16     negCycle := TryBackProp(G, E, Q, global, loc)     // If TryBackProp finds an SRN cycle, then return it
17     if negCycle ≠ ∅ then return negCycle
18     if loc.unstartedUCs ≠ ∅ then                      // Process unstarted UC-edges
19         foreach (E_X, X) ∈ loc.unstartedUCs do
20             global.intBy[E] := (E_X, loc.path[X])     // Record interrupting UC edge E_X and path from X to A
21             negCycle := RulBackProp(G, E_X, global)   // If processing E_X yields an SRN cycle, return it
22             if negCycle ≠ ∅ then return negCycle
23         global.intBy[E] := ⊥                          // Record that all unstarted UC edges (found so far) have been successfully processed
24         Q.clear()                                     // Prepare Q for next iteration of the while loop
25         foreach (E_X, X) ∈ loc.unstartedUCs do
26             Q.insert(X, loc.dist[X] + global.pf[X])
27     else continue? := ⊥                               // Back-prop. from E completed
28 if loc.ccLoop then                                    // CC-loop found; must initiate forward propagation
29     (X, P_X) := FwdPropNDC(G, C, Δ_C, loc, global.pf)
30     if (X, P_X) ≠ ∅ then return (A, c:x, C) + P_X + loc.path[X]   // If (A, c:x, C) can be reduced away, then return SRN cycle
31 foreach X ∈ T \{C} do                                 // Generate bypass edges using U_lp rule
32     δ_xc := loc.dist[X]                               // δ_xc = ∞ means node not reachable
33     if Δ_C ≤ δ_xc < ∞ then
34         G.insertOrdEdge(X, δ_xc − y, A)
35         global.edgeAnnotation.put((X, A), loc.path[X])   // Record path for bypass edge in edgeAnnotation hash
36         addedEdges? := ⊤
   /* If inserted any bypass edges, then need to update the potential function */
37 if addedEdges? then (global.pf, negCycle) := UpdatePotFn(G, A, global.pf)
38 if global.pf == ⊥ then return negCycle
39 global.status[E] := done
40 return ∅                                              // Processing of E successfully completed
```

Pseudocode for the `RulBackProp`algorithm is in Algorithm 9. It processes a single UC edge $\mathbf{E} = (C, C{:}{-}y, A)$ while integrating the recursive processing of any interrupting UC edges. At Line 2, it checks whether the processing of $\mathbf{E}$ has already been started, but not yet completed, which implies a negative cycle of interruptions. In this case, `RulBackProp`calls `AccNegCycle` (Algorithm 10) to collect the relevant path information accumulated in the `global.intBy` vector, which is then returned as a compact representation of an SRN cycle. (More will be said about how the information in `global.intBy` is generated.) At Line 3, if $\mathbf{E}$ has already been successfully processed, it immediately returns $\top$.

Algorithm 10: The `AccNegCycle` algorithm (new).

Input: `global.intBy`, vector recording a cycle of interruptions; $\mathbf{E} \in \mathcal{E}_u$, a UC edge in the cycle
Output: *negCycle*, an SRN cycle containing $\mathbf{E}$

```
1 negCycle := ⟨⟩
2 (E', P') := global.intBy[E]                    // P' is path used to generate E'
3 while E' ≠ E do
4     negCycle := P' + negCycle                  // Accumulate P' into cycle
5     (E', P') := global.intBy[E']               // Fetch next interrupter
6 return P' + negCycle
```

At Lines 4-10, `RulBackProp`prepares to process a UC edge $\mathbf{E} = (C, C{:}{-}y, A)$. As in `RUL2021`, `ccLoop` is a flag used to signal the discovery of a *CC loop*; and `dist` records, for each encountered timepoint X, the distance from X to C in the LO-graph. A new field, `path`, records the paths from each X to A (via C). (We use $\langle\rangle$ to denote an empty path.) Back-propagation from C is governed by a priority queue $\mathcal{Q}$, initialized at Lines 11-14 to include each X connected to C by an edge. For each such X, `loc.path`$[X]$ records the two-edge path from X to C to A. (We use $+$ as a concatenation operator that can be applied to edges or paths. For example, if e_1 is an edge, and π_1 and π_2 are paths, then $\pi_1 + e_1 + \pi_2$ represents their concatenation into a single path.)

In each iteration of the `while` loop (Lines 15-27), `RulBackProp`either starts *or resumes* the processing of $\mathbf{E}$, first (at Line 16) by calling `TryBackProp` (Algorithm 11). `TryBackProp` (described later) back-propagates along LO-edges, but does *not* generate or insert any bypass edges. Instead, it simply collects the relevant distance *and path* information, while also keeping track of whether it encountered any unstarted (i.e., interrupting) UC edges or *CC loops.* At Line 17, `RulBackProp`checks whether `TryBackProp` found an SRN cycle, in which case `RulBackProp`returns that cycle. Otherwise, at Line 18, `RulBackProp`checks whether `TryBackProp` encountered any interrupting UC edges. If so, for each interrupting UC edge $\mathbf{E}_X$ (Lines 19-22), it uses `global.intBy` [$\mathbf{E}$] to record the interruption and then attempts to recursively process $\mathbf{E}_X$. If all interrupting UC edges are successfully processed, it clears the `global.intBy` [$\mathbf{E}$] entry (Line 23) and prepares for the next iteration of the `while` loop by re-initializing the priority queue (Lines 24-26) so that processing $\mathbf{E}$ can be resumed, starting from the activation timepoints of the no-longer-interrupting UC edges.

Once all back-propagation from $\mathbf{E}$ is done, `RulBackProp`checks, at Line 28, whether any *CC loops* were encountered. If so, it calls `FwdPropNDC` to carry out a separate forward propagation from C along LO-edges, checking whether any LO-path, $\mathcal{P}_{CX}$, from C to some X, can be used to bypass the LC edge $e = (A, c{:}x, C)$. If so, there must be an SRN cycle, $e + \mathcal{P}_{CX} +$ `loc.path`$[X]$, where `loc.path`$[X]$ is the LO-path from X to A obtained by the earlier back-propagation from C [36]. Hence, `FwdPropNDC` returns $(X, \mathcal{P}_{CX})$. For the STNU on the left of Fig. 4, $\mathcal{P}_{CX}$ is $(C, 1, W) + (W, -3, X)$ and `loc.path`$[X]$ is $(X, 4, C) + (C, C{:}{-}9, A)$.

If forward propagation fails to find an SRN cycle, then `RulBackProp`finally uses the information in `loc.dist` to generate edges that bypass the UC edge $\mathbf{E}$ (Lines 31-36). These are the only edges that `findSRNC` actually inserts into the STNU. For each bypass edge $(X, \delta_{xc} - y, A)$, the corresponding path that has been accumulated in `loc.path`$[X]$ is recorded in the `global.edgeAnnotation` hash table (Line 35). (As discussed below, it is `TryBackProp` that accumulates the path information in `loc.path`$[X]$.) If any bypass edges are inserted, then `RulBackProp`(at Line 37) calls `UpdatePotFn` (Algorithm 13, discussed later) to update the potential function for the LO-graph, whence the processing of $\mathbf{E}$ is completed (Line 39).

`TryBackProp`. Pseudocode for `TryBackProp` is given as Algorithm 11. Like its predecessor `RUL2021_TryBackProp` (Algorithm 3), the `TryBackProp` algorithm propagates backward from a UC edge $\mathbf{E} = (C, C{:}{-}y, A)$, using a priority queue and potential function to explore shortest LO-paths. And, similarly, each iteration of its `while` loop focuses on a timepoint X for which `loc.dist`$[X] < \Delta_C$ (Line 9) and considers four cases (Lines 10-19). The main difference is in how it responds to Cases 3 and 4.

In Case 3 (Lines 12-14), where back propagation has discovered a cycle of interrupting UC edges, `TryBackProp` records the interruption of $\mathbf{E}$ by $\mathbf{E}_X$ along with the corresponding *path* from X to A that has been accumulated in `loc.path`$[X]$. It then calls `AccNegCycle` (Algorithm 10), which does a straightforward recursive traversal of the information in `global.intBy` to generate a compact representation of the SRN cycle.

In Case 4 (Lines 15-19), where back propagation continues past X, `TryBackProp` accumulates additional path information using the *edge*/distance pairs (e, δ_{wc}) returned by `ApplyRL` (Algorithm 15), instead of the *timepoint*/distance pairs (W, δ_{wc}) returned by `RUL2021_ApplyRL` (Algorithm 7). At Line 19, it uses the (ordinary or lower-case) edge e to update the path information from W to A stored in `loc.path`$[W]$.

`FwdPropNDC`. The `FwdPropNDC` algorithm (Algorithm 12) propagates *forward* from C along LO-edges checking whether there is a negative-length path from C to some X that can be used to bypass the LC edge $(A, c{:}x, C)$. It is the same as in `RUL2021`, except that it accumulates path information in a vector called `fwdPath`. At Lines 2-3, a priority queue is initialized to contain just C, with `fwdPath`$[C] = \langle\rangle$. The priority queue uses the same potential function as `TryBackProp` to effectively re-weight the LO-edges. As each timepoint X is popped from the queue (Line 5), the distance from X to C that

Algorithm 11: The `TryBackProp` algorithm.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; $\mathbf{E} = (C, C{:}{-}y, A) \in \mathcal{E}_u$; $\mathcal{Q}$, a priority queue; `global`, a *global* struct; `loc`, a *local* struct
Output: *negCycle*, an SRN cycle; or $\emptyset$ if no SRN cycle found.
Side Effect: Propagates backward from C along shortest LO-paths, updating contents of `loc` and `global`, but no change to $\mathcal{G}$

```
1  f := global.pf
2  st := global.status
3  loc.unstartedUCs := {}
4  while !(Q.empty()) do
5      (X, key_x) := Q.extractMinNode()
6      δ_xc := key_x − f(X)
7      loc.dist[X] := δ_xc
8      E_X := G.UCEdgeFromATP(X)
9      if δ_xc < Δ_C then
10         if X ≡ C then loc.ccLoop := ⊤                                  // Case 1: CC loop of length δ_xc < Δ_C
11         else if E_X and st[E_X] == notYet then loc.unstartedUCs.add((E_X, X))   // Case 2: Interrupting UC edge
12         else if E_X and st[E_X] == started then                        // Case 3: Cycle of interruptions: not DC
13             global.intBy[E] := (E_X, loc.path[X])                      // Record interrupting UC edge and associated path
14             return AccNegCycle(global.intBy, E_X)                      // AccNegCycle computes SRN cycle determined by interruptions
15         else foreach (e, δ_wc) ∈ ApplyRL(G, X, Δ_C, δ_xc) do           // Case 4: Continue back-propagation along LO-edges
16             newKey := δ_wc + f(W)
17             if δ_wc < loc.dist[W] and (W ∉ Q or newKey < Q.key(W)) then
18                 Q.insertOrDecreaseKey(W, newKey)
19                 loc.path[W] := e + loc.path[X]                         // Accumulate new path from W to C
20 return ∅
```

Algorithm 12: The `FwdPropNDC` algorithm.

Input: $\mathcal{G}$, an STNU graph; $C \in \mathcal{T}_C$; $\Delta_C = y - x$; `loc`, *local* struct; f, potential function.
Output: $(X, \mathcal{P}_{CX})$, if path $\mathcal{P}_{CX}$ can be used to reduce away the LC edge $(A, c{:}x, C)$; else $\emptyset$

```
1  fwdPath := {⟨⟩, ..., ⟨⟩}                                  // For each X, fwdPath[X] is an LO-path from C to X
2  Q := new priority queue                                   // Key key(X) = d(C, X) − h(X)
3  Q.insert(C, −f(C))                                        // Queue initially contains only C
4  while Q ≠ ∅ do
5      (X, key(X)) := Q.extractMinNode()
6      d(C, X) := key(X) + f(X)                              // Distance from C to X in G_ℓo
7      if loc.dist[X] < Δ_C then                             // If distance from X to C < Δ_C
           /* Check if the path CX can reduce-away the LC-edge */
8          if d(C, X) < 0 then return (X, fwdPath[X])
9      foreach (X, δ_xy, Y) ∈ E_ℓ ∪ E_o do                   // Iterate over LO-edges emanating from X
10         newKey := d(C, X) + δ_xy − f(Y)
11         if Y ∉ Q or newKey < Q.key(Y) then
12             Q.insertOrDecreaseKey(Y, newKey)
13             fwdPath[Y] := fwdPath[X] + (X, δ_xy, Y)
14 return ∅                                                  // Was unable to reduce-away the LC-edge
```

was determined during back-propagation and stored in `loc.dist`$[X]$ is compared to Δ_C. (Generating an edge to bypass the LC edge using the path from C to X will only create an SRN cycle if `dist`$[X] < \Delta_C$ [36].) If `dist`$[X] < \Delta_C$ and $d(C, X) < 0$ (i.e., an appropriate negative-length path has been found), then `FwdPropNDC` terminates, returning $(X,$ `fwdPath`$[X])$ (Line 8). Otherwise, forward propagation continues from X, accumulating relevant path information (Lines 9-13). If the queue is exhausted without finding a way to bypass the LC edge, `FwdPropNDC` returns $\emptyset$ (Line 14).

`UpdatePotFn` *and* `UpdateVal`. Pseudocode for the `UpdatePotFn` and `UpdateVal` functions is given as Algorithms 13 and 14. These functions are the same as in `RUL2021` except that path information is accumulated (Algorithm 14, Line 5) so that if back-propagation ever cycles all the way back to A, the implied SRN cycle can be returned (Algorithm 14, Line 3).

Computational complexity. The `findSRNC` algorithm performs more operations than `RUL2021`, mostly by accumulating path information during propagation. The most time-consuming operation is prepending an edge onto the front of an existing path, which happens at most once per edge visited. Since the prepending operation (using $+$) can be done in constant time, the worst-case complexity of `findSRNC` is the same as that of `RUL2021`: $O(mn + k^2 n + kn \log n)$.

Algorithm 13: The `UpdatePotFn` algorithm.

Input: $\mathcal{G}$, an STNU graph; A, an activation timepoint; f, a potential function for $\mathcal{G}_{\ell o}$, excluding edges ending at A
Output: $(f', negCycle)$, where f' is either the updated potential function for $\mathcal{G}_{\ell o}$ (*including* edges terminating at A); or $\perp$, the latter indicating that *negCycle* is a negative cycle

```
1  f' := copy_vector(f)
2  path := [⟨⟩, ..., ⟨⟩]
3  Q := new empty priority queue
4  Q.insert(A, 0)                                   // Initialize queue for back-propagation from A
5  while ¬Q.empty() do
6      (V, keyV) := Q.extractMinNode()
7      foreach (U, δ, V) ∈ E_o do                   // Back-propagate along ordinary edges ending at V
8          negCycle := UpdateVal((U, δ, V), f, f', Q, path)
9          if negCycle ≠ ∅ then return (⊥, negCycle)
10     if V ∈ T_C then                              // If V is contingent, back-propagate along LC edge (A_V, v:x_V, V)
11         (A_V, x_V, y_V, V) := contingent link for V
12         negCycle := UpdateVal((A_V, x_V, V), f, f', Q, path)
13         if negCycle ≠ ∅ then return (⊥, negCycle)
14 return (f', ∅)
```

Algorithm 14: The `UpdateVal` algorithm.

Input: (U, δ, V), an edge; f and f', original and updated potential functions; $\mathcal{Q}$, priority queue; and *path*, a vector of path info
Output: *negCycle*, an SRN cycle; or $\emptyset$ if f' was successfully updated to satisfy (U, δ, V)
Side Effect: Modifies $\mathcal{Q}$, f' and *path*

```
1  if f'(U) < f'(V) − δ then
2      f'(U) := f'(V) − δ
       /* If back propagation has cycled back to A, return the SRN cycle */
3      if Q.state(U) == alreadyPopped then return (U, δ, V) + path[V]
4      Q.insertOrDecreaseKey(U, f(U) − f'(U))
5      path[U] := (U, δ, V) + path[V]
6  return ∅
```

Algorithm 15: The `ApplyRL` algorithm.

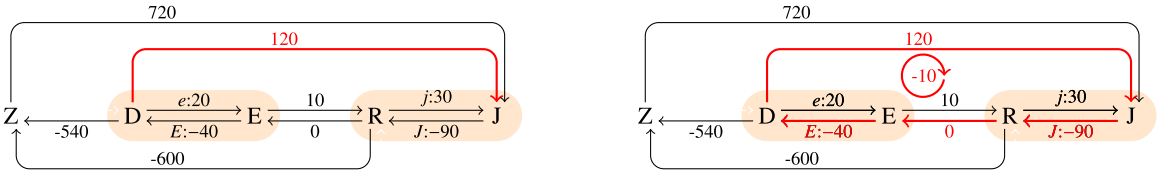
Input: $\mathcal{G}$, an STNU graph; $V \in \mathcal{T}$; $\Delta_C = y - x$; and δ_{vc}, the distance from V to C in the LO-graph
Output: A list of edge-distance pairs, (e, δ_{wc}), each representing the result of applying the R or L rule to an LO-edge e and (V, δ_{vc}, C), where $\delta_{wc} = |e| + \delta_{vc}$

```
1  pairs := {}
2  if δ_vc < Δ_C then                               // Application condition for R and L rules
3      if V ∈ T_C then pairs.add(((A_V, v:x_V, V), x_V + δ_vc))   // Apply the L rule to (A_V, v:x_V, V) and (V, δ_vc, C)
4      else foreach (W, δ_wv, V) ∈ E_o do pairs.add(((W, δ_wv, V), δ_wv + δ_vc))   // Apply the R rule to each (W, δ_wv, V) and (V, δ_vc, C)
5  return pairs
```

Regarding the extra space requirements of `findSRNC`, the most costly is the space needed by `TryBackProp` for accumulating path information in the `loc.path` structures. `TryBackProp` is called at most $2k$ times [12,36]. Each call explores at most $(m+nk)$ edges. (`findSRNC` inserts at most nk edges overall.) Each edge exploration involves prepending an existing path with an edge, which uses only constant space. So the overall space complexity across *all* calls to `TryBackProp` is $O(mk + k^2 n)$. Similar remarks apply to `FwdPropNDC` and `UpdatePotFn`.

The `edgeAnnotation` hash table has at most nk entries: one for each bypass edge. Each entry is a *pointer* to a `loc.path` entry; so the total space required is $O(nk)$. The *compact* SRN cycle generated by `AccNegCycle` is the concatenation of at most k paths, each with at most n edges, for a total of at most nk edges, which is dominated by the $O(mk+k^2n)$ space discussed above. This compact cycle, together with the information in the `edgeAnnotation` hash table, avoids redundantly storing repeated structures. Thus, it uses polynomial space to *implicitly* represent a cycle that, if fully expanded, might have exponentially many edges. Similar remarks apply to the cycles returned by `FwdPropNDC` and `UpdatePotFn`.

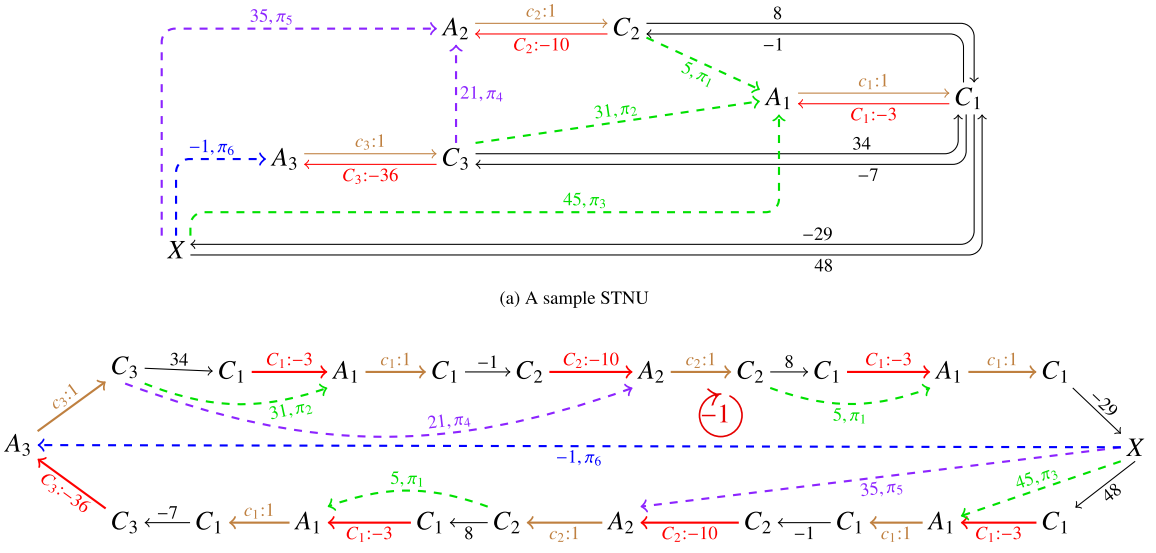
Over-constrained STNU repair scenario. Fig. 6a shows the STNU for the Repair Scenario, except that the constraint from D to J has been tightened to 120. This tightening makes the STNU non-DC. The `findSRNC` algorithm determines the presence of the negative cycle $\langle D, J, R, E, D\rangle$ of length -10, as shown in Fig. 6b. This SRN cycle can be repaired by *increasing* the length of the ordinary edge $(D, 120, J)$ by 10. Alternatively, it can be repaired by *decreasing* the upper bounds of one or both of the contingent links by a total of 10 minutes. It is not permitted to increase the length of the ordinary edge $(R, 0, E)$



(a) STNU for the Repair Scenario, but with a modified constraint

(b) Same STNU but with SRN cycle identified by `findSRNC`

Fig. 6. Repair-Scenario STNU with a SRN cycle.



(a) A sample STNU

(b) The magic loop of order 3 hiding within Figure 7a

Fig. 7. A sample STNU and the magic loop hiding within it.

because it represents a necessary precedence constraint, $E \leq R$. (Since the job must be done at the client's house, the job cannot start until the technicians arrive at the client's house.)

Magic loop example. Hunsberger [33] identified a family of STNUs in which the only SRN cycle, called a *magic loop,* has an *exponential* number of edges. Since each STNU has at most $n^2 + 2k$ distinct edges, magic loops necessarily contain a large number of repeated edges. In particular, a magic loop *of order k* has k contingent links, but $3(2^k) - 2$ edges. For example, consider the STNU depicted in Fig. 7a, which has three (brown) LC edges, $\mathbf{e}_1 = (A_1, c_1{:}1, C_1)$, $\mathbf{e}_2 = (A_2, c_2{:}1, C_2)$, and $\mathbf{e}_3 = (A_3, c_3{:}1, C_3)$, and three (red) UC edges, $\mathbf{E}_1 = (C_1, C_1{:}{-3}, A_1)$, $\mathbf{E}_2 = (C_2, C_2{:}{-10}, A_2)$, and $\mathbf{E}_3 = (C_3, C_3{:}{-36}, A_3)$. The bypass edges generated by `findSRNC` are dashed: those bypassing $\mathbf{E}_1$ in green, $\mathbf{E}_2$ in purple, and $\mathbf{E}_3$ in blue. Each bypass edge is also annotated by a path, where: $\pi_1 = (C_2, 8, C_1) + \mathbf{E}_1$; $\pi_2 = (C_3, 34, C_1) + \mathbf{E}_1$; $\pi_3 = (X, 48, C_1) + \mathbf{E}_1$; $\pi_4 = \pi_2 + \mathbf{e}_1 + (C_1, -1, C_2) + \mathbf{E}_2$; $\pi_5 = \pi_3 + \mathbf{e}_1 + (C_1, -1, C_2) + \mathbf{E}_2$; and $\pi_6 = \pi_5 + \mathbf{e}_2 + \pi_1 + \mathbf{e}_1 + (C_1, -7, C_3) + \mathbf{E}_3$. The magic loop for this STNU is depicted in Fig. 7b. It has 22 edges: 8 ordinary edges, 7 LC edges, and 7 UC edges. $\mathbf{E}_1$ and $\mathbf{e}_1$ appear four times each; others twice each. After all UC edges have been processed, `UpdatePotFn` finds a negative cycle in the LO-graph: $\pi_6 + \mathbf{e}_3 + \pi_4 + \mathbf{e}_2 + \pi_1 + \mathbf{e}_1 + (C_1, -29, X)$. This information is compactly stored in the cycle returned by `findSRNC`. For higher-order magic loops, the number of edges grows exponentially, but the space used by `findSRNC` is bounded by $mk + nk^2$.

6. From STN dispatchability to (e)STNU dispatchability

This section begins by summarizing relevant definitions and results from the comprehensive literature on the dispatchability of Simple Temporal Networks. It then presents initial approaches to the foundations of dispatchability for Simple Temporal Networks with Uncertainty, which requires extending STNUs to include conditional *wait* constraints, resulting in Extended Simple Temporal Networks with Uncertainty (ESTNUs).

6.1. STN dispatchability

Although checking the consistency of an STN and finding a solution for it can be done in polynomial time, fixing a solution in advance of execution undermines the inherent flexibility of the STN representation. Instead, it can be desirable to preserve as much flexibility as possible until actions are actually performed (e.g., to enable reacting to unanticipated events), while minimizing the need for real-time computation. Toward that end, Tsamardinos et al. [72] first specified a real-time execution algorithm for STNs. Their real-time execution algorithm provides maximum flexibility during execution by maintaining *time windows* for each timepoint, while requiring minimal real-time computation by only *locally* propagating the effects of each real-time execution event, $X = t$ (i.e., propagating only to X's *neighbors*; that is, timepoints connected to X by a single edge). They then defined an STN to be *dispatchable* if every run of that algorithm on that STN was guaranteed to generate a solution no matter how the flexibility afforded by the RTE algorithm is exploited in real time.

Algorithm 16: RTE: real-time execution for STNs.

Input: $(\mathcal{T}, \mathcal{C})$, an STN with graph $(\mathcal{T}, \mathcal{E})$
Output: A function, $f : \mathcal{T} \to [0, \infty)$ or `fail`

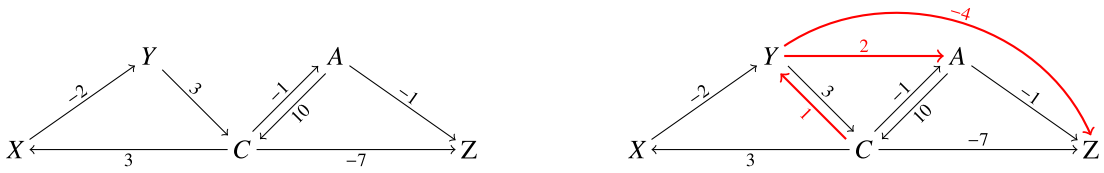
1 **foreach** $X \in \mathcal{T}$ **do** $\texttt{Win}(X) = [0, \infty)$ // Initialize time windows
2 $\mathcal{U} := \mathcal{T}$; $\texttt{now} = 0$ // All timepoints initially unexecuted; current time = 0 (arbitrary)
3 $\texttt{Enabs} := \{X \in \mathcal{T} \mid X \text{ has no outgoing negative edges}\}$ // Initialize set of enabled timepoints
4 **while** $\mathcal{U} \neq \{\}$ **do** // While some timepoints not yet executed
5 **if** $\texttt{Enabs} = \emptyset$ **then return** `fail` // If no timepoints enabled, then `fail`
6 $\ell := \min\{lb(W) \mid W \in \texttt{Enabs}\}$; $u := \min\{ub(W) \mid W \in \texttt{Enabs}\}$ // $[\ell, u]$ = maximal time window for next execution event
7 **if** $[\ell, u] \cap [\texttt{now}, \infty) = \emptyset$ **then return** `fail` // If maximal time window for next event is in the past, then `fail`
8 Select *any* $X \in \texttt{Enabs}$ such that $\texttt{Win}(X) \cap [\texttt{now}, u] \neq \emptyset$ // Select an enabled timepoint whose time window is "open"
9 Select *any* $t \in \texttt{Win}(X) \cap [\texttt{now}, u]$ // Select any time from X's time window, respecting the maximal time window
10 $\mathcal{U} := \mathcal{U} \backslash \{X\}$; $f(X) := t$; $\texttt{now} := t$ // Execute X at time t
11 **foreach** $(X, \delta_{xy}, Y) \in \mathcal{E}$ **do** $\texttt{Win}(Y) := \texttt{Win}(Y) \cap (-\infty, t + \delta_{xy}]$ // Propagate upper bounds along edges leaving X
12 **foreach** $(W, \delta_{wx}, X) \in \mathcal{E}$ **do** $\texttt{Win}(W) := \texttt{Win}(W) \cap [t - \delta_{wx}, \infty)$ // Propagate lower bounds along edges incoming to X
13 $\texttt{Enabs} := \{Y \in \mathcal{U} \mid \text{all negative edges from } Y \text{ terminate at timepoints not in } \mathcal{U}\}$ // Update set of enabled timepoints
14 **return** f

Consider the Real-Time Execution (RTE) algorithm for STNs given in Algorithm 16 [60].[9] It provides maximum flexibility by maintaining for each timepoint X a *time window,* $\texttt{Win}(X)$ (initially $[0, \infty)$, Line 1), and providing maximal freedom for which timepoint to execute next and when to execute it (Lines 6-9). To minimize real-time computation, the effects of each execution decision, $X = t$ (represented in the pseudocode by setting $f(X) = t$ at Line 10) are propagated only *locally,* to the *neighbors* of X in the STN graph, with upper bounds propagated along outgoing edges (Line 11) and lower bounds propagated along incoming edges (Line 12).

After initializing the time windows (Line 1), the RTE algorithm initializes the *current time* `now` to 0 and the set $\mathcal{U}$ of *unexecuted* timepoints to $\mathcal{T}$ (Line 2); and then the set of *enabled* timepoints to those having no outgoing negative edges (Line 3). (A timepoint Y is *enabled* for execution if it is *not* constrained to occur *after* any *unexecuted* timepoint—equivalently, if there are no *negative* edges from Y to any *unexecuted* timepoint.) Each iteration of the `while` loop (Lines 4-13) begins by computing the interval $[\ell, u]$, where ℓ is the minimum lower bound of the time windows among the enabled timepoints (i.e., the earliest time at which something *could* happen) and u is the minimum upper bound among those same time windows (i.e., the deadline by which something *must* happen) (Line 6).[10] The algorithm fails if that interval does not include times at or after `now` (Line 7). Next (Line 8), it selects *any* one of the enabled timepoints X whose time window $\texttt{Win}(X)$ has a non-empty intersection with $[\texttt{now}, u]$, and then (Line 9) selects *any* time $t \in \texttt{Win}(X) \cap [\texttt{now}, u]$ at which to execute it. (If $[\ell, u] \cap [\texttt{now}, \infty)$ is non-empty, then there must be such an X.) After assigning X to t (Line 10), it then propagates the effects of that assignment to X's neighbors in the STN graph, propagating upper bounds along edges leaving X (Line 11) and lower bounds along edges incoming to X (Line 12). Finally, it updates the set of enabled timepoints (Line 13) in preparation for the next iteration.

The RTE algorithm for STNs provides maximal flexibility in that *any* solution to a consistent STN can be generated by an appropriate sequence of choices at Lines 8-9. In addition, it requires minimal computation by performing only *local* propagation (at Lines 11-12). However, it does *not* provide a constraint-satisfaction guarantee for *all* runs of the algorithm on consistent STNs. For example, consider the sample STN shown in Fig. 8a, modified from Hunsberger and Posenato [41]. Table 3 shows a sample run of the RTE algorithm on this STN which fails to generate a solution. The fact that the RTE algorithm can fail for consistent STNs motivates the work on STN *dispatchability,* as follows.

[9] Muscettola et al. [60] refer to their algorithm as either the *Time Dispatching Algorithm* (TDA) or the *Dispatching Execution Controller* (DEC). The RTE algorithm presented here is equivalent, although for convenience it is organized somewhat differently and uses some different notation.
[10] In Algorithm 16, $\texttt{lb}(X)$ and $\texttt{ub}(X)$ respectively denote the lower and upper bounds from X's time window, $\texttt{Win}(X)$.



(a) A sample of STN (modified from Hunsberger and Posenato [41])

(b) An equivalent *dispatchable* STN

Fig. 8. A sample STN and an equivalent *dispatchable* STN.

Table 3
A sample run of the RTE algorithm that fails on the *consistent* STN from Fig. 8a.

Iteration	`Enabs`	`Win(Z)`	`Win(`A`)`	`Win(`C`)`	`Win(`X`)`	`Win(`Y`)`	$[\ell, u]$	`now`	Execution
Initialization	$\{Z, Y\}$	$[0, \infty]$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	0	$Z := 0$
1	$\{A, Y\}$	–	$[1, \infty)$	$[7, \infty)$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	0	$Y := 2$
2	$\{A, X\}$	–	$[1, \infty)$	**[7, 5]**	$[4, \infty)$	–	$[1, \infty)$	2	$A := 8$
3	$\{C, X\}$	–	–	**[9, 5]**	$[4, \infty)$	–	**[4, 5]**	**8**	**`fail`**

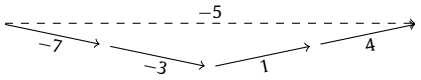


Fig. 9. A sample *vee-path* that dominates a direct edge.

Definition 1 *(STN Dispatchability, Muscettola et al. [60]).* An STN $\mathcal{S} = (\mathcal{T}, \mathcal{C})$ is *dispatchable* if every run of the RTE algorithm (Algorithm 16) on the corresponding STN graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ necessarily generates a solution for $\mathcal{S}$.

Muscettola et al. [60] showed that for every consistent STN, its APSP graph is necessarily dispatchable, but its $O(n^2)$ edges cancel the benefits of *local* propagation. Their $O(n^3)$-time *edge-filtering* algorithm computes an equivalent dispatchable STN having a *minimal* number of edges by first constructing the APSP graph and then removing *dominated* edges (i.e., edges not needed for dispatchability). A faster $O(mn + n^2 \log n)$-time algorithm accumulates the undominated edges without first building the APSP graph [72]. In both algorithms, having the *minimal* number of edges in the dispatchable output is important because it minimizes the amount of local propagation required during execution.

Morris [55] later found a graphical characterization of STN dispatchability in terms of *vee-paths*.

Definition 2 *(Vee-paths [55]).* A *vee-path* is a path comprising zero or more negative edges followed by zero or more non-negative edges. A vee-path from X to Y that is also a shortest path from X to Y is called a *shortest vee-path* (SVP).

Fig. 9 shows a sample vee-path from X to Y that dominates the (dashed) direct edge from X to Y. For this vee-path, the enablement condition in the RTE algorithm (Algorithm 16, Line 8) ensures that the algorithm will execute B before A, and A before X; hence, local propagation ensures the satisfaction of the edges $(X, -7, A)$ and $(A, -3, B)$. On the other side, if RTE executes C before B, then the edge $(B, 1, C)$ is automatically satisfied; otherwise, local propagation ensures its satisfaction. Similarly, RTE necessarily satisfies the edge $(C, 4, D)$. Since RTE satisfies all the edges in the vee-path, it also satisfies the direct edge $(X, -5, Y)$. Hence that edge is not needed to ensure dispatchability.

Theorem 1 *(Morris [55]). An STN is dispatchable if and only if for each path from any X to any Y in the STN graph, there is a* shortest vee-path *from X to Y.*

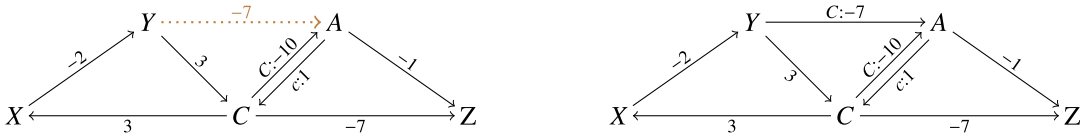
Fig. 8b shows a dispatchable STN that is equivalent to the STN from Fig. 8a (new edges are thick and red). It is easy to check that each path has a corresponding SVP. Table 4, also modified from Hunsberger and Posenato [41], shows a sample run of the RTE algorithm on this dispatchable STN, which necessarily generates a solution.

Definition 3 *(Vee-path completeness).* An STN graph $\mathcal{G}$ is *vee-path complete* if for every X and Y for which there is a path from X to Y in $\mathcal{G}$ there is a shortest vee-path from X to Y.

With this definition, Theorem 1 can be restated as: An STN is dispatchable iff it is vee-path complete.

Table 4
A sample run of the RTE algorithm on the *dispatchable* STN from Fig. 8 (modified from Hunsberger and Posenato [41]).

Iteration	Enabs	Win(Z)	Win(A)	Win(C)	Win(X)	Win(Y)	$[\ell, u]$	now	Execution
Initialization	{Z}	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	$[0, \infty)$	0	$Z := 0$
1	$\{A, Y\}$	–	$[1, \infty)$	$[7, \infty)$	$[0, \infty)$	$[4, \infty)$	$[1, \infty)$	0	$A := 8$
2	$\{C, Y\}$	–	–	[9, 18]	$[0, \infty)$	$[4, \infty)$	[4, 18]	8	$C := 15$
3	$\{Y\}$	–	–	–	[0, 18]	[4, 16]	[4, 16]	15	$Y := 16$
4	$\{X\}$	–	–	–	[18, 18]	–	[18, 18]	16	$X := 18$



(a) A sample STNU (modified from Hunsberger and Posenato [41])

(b) An ESTNU with a wait edge $(Y, C{:}-7, A)$

Fig. 10. A sample STNU graph and a similar ESTNU graph.

RTE complexity. With appropriate data structures, the RTE algorithm can be implemented to run in a total of $O(n^2)$ worst-case time, while allowing for maximum flexibility in the selection of each timepoint to execute next and the time at which to execute it. First, whenever a timepoint is executed, the time windows of its neighbors are updated. If the queues are realized using strict Fibonacci heaps [10], each update requires $O(1)$ time. Since each update corresponds to an edge in the graph, the total time for doing *all* of those updates is $O(m)$. Second, the set of enabled timepoints can be implemented by keeping, for each timepoint, a count of its outgoing negative edges. Whenever a negative edge is processed, the count for the source of that edge is decremented. When the count for a given timepoint reaches 0, that timepoint becomes enabled. Since there are at most m negative edges, the total cost of managing enabled timepoints is $O(m)$. Third, to compute the values of ℓ and u, it suffices to maintain two minimum priority queues [19], one for ℓ and one for u. When a timepoint X becomes enabled, it is inserted into both queues using its $\texttt{lb}(X)$ and $\texttt{ub}(X)$ values as keys. To compute the current values of ℓ and u requires only "peeking" at the current minimum value in each queue. Although *executed* timepoints should be removed from both queues because they are no longer relevant for determining ℓ and u, they need not be extracted immediately upon execution, but instead can be extracted lazily, as follows. Whenever a "peek" reveals a bound associated with an already-executed timepoint, that timepoint can be extracted from the queue at that time, at a cost of $O(\log n)$. Subsequent peek/extractions can be done until a peek reveals a value based on a not-yet-executed timepoint. In this way, each timepoint is inserted and extracted exactly once, for a total cost of $O(n \log n)$. Since each peek can be done in constant time, the cost of peeks that lead to a timepoint extraction can be ignored; and the cost of peeks that generate ℓ or u values, which happens once per iteration, is $O(n)$ overall. In addition, each update to the time window of a timepoint in one of the queues requires doing a "decrease key" operation which, with a strict Fibonacci heap, can be done in $O(1)$ time, for a total cost of $O(m)$. Therefore, the overall cost for managing the two queues is $O(m + n \log n)$.

For full flexibility, $O(n)$ worst-case time is required for selecting the timepoint X to execute next, which drives the overall $O(n^2)$ worst-case time. Selecting the time t at which to execute X, if done randomly, can be done in $O(1)$ time. However, applications may have domain-specific criteria that make the selections of X and t more time-consuming.

6.2. Toward dispatchability for (e)STNUs

As with STNs, it is important to preserve maximal flexibility for STNUs during execution, while requiring minimal real-time computation. Hence, it is important to define dispatchability for STNUs. Unfortunately, a DC STNU need not have an equivalent dispatchable *STNU.* For example, consider the sample STNU shown in Fig. 10a, ignoring the dotted edge for now. If A executes at time 0, then C may execute at any time in $[1, 10]$. To ensure that the constraint $C - Y \leq 3$ (i.e., $C \leq Y + 3$) is satisfied, it follows that as long as C remains unexecuted, Y cannot execute before time 7. But enforcing this constraint by inserting the edge $(Y, -7, A)$, shown as dotted in the figure, would result in an STNU that not only is not equivalent to the original, but is not even dynamically controllable. (With the dotted edge, the path $(A, c{:}1, C, 3, X, -2, Y, -7, A)$ would constitute a negative cycle in the LO-graph.) Instead, as described later in this section, to create equivalent dispatchable networks for arbitrary DC STNUs, it is necessary to increase the expressivity of STNUs by including a new kind of edge, called a *wait edge,* that represents a *conditional* constraint.

Anticipating this, Morris [54] defined an *Extended STNU* (ESTNU) to be an STNU augmented with wait edges. He then *defined* an ESTNU to be dispatchable if all of its projections were dispatchable (as STNs); and then argued that a dispatchable ESTNU would necessarily be successfully executed by an appropriate real-time execution algorithm, but did not provide detailed proofs. Later, Hunsberger and Posenato [41] formalized a real-time execution algorithm for ESTNUs, called RTE*, and then proved that for a dispatchable ESTNU (according to Morris' definition), every run of RTE* on that ESTNU would neces-

sarily satisfy all of its constraints. By doing so, they formally confirmed that every DC STNU has an equivalent dispatchable ESTNU.[11]

Wait edges. Many of the early DC-checking algorithms generate a new kind of edge, called a *wait edge,* that represents a *conditional* constraint [53]. A typical wait edge may be notated as $(V, C{:}{-}w, A)$, where A and C are the activation and contingent timepoints for a contingent link. Such a wait can be glossed as *"If the contingent timepoint C has not yet executed, then V must wait until at least w after the activation timepoint A."* If C happens to execute late, then V must wait until at least w after A; but if C executes early, then the wait is automatically satisfied and V can be executed at any time. Although the notation for wait edges is similar to that of UC edges, they are distinguished by the fact that V is distinct from the contingent timepoint C.

In general, it is not necessary to generate wait edges to determine the DC property (e.g., as seen in Morris' 2014 algorithm [54], the RUL$^-$ algorithm [12], and the `RUL2021` algorithm discussed in Section 4). However, waits turn out to be necessary for enforcing the dispatchability of STNUs. Thus, Morris [54] defined an ESTNU to be an STNU augmented with a set of conditional wait constraints, and an ESTNU graph to include a corresponding set $\mathcal{E}_w$ of wait edges. Fig. 10b shows a sample ESTNU graph. It is the same as that STNU from Fig. 1b, except that it has a wait edge $(Y, C{:}{-}7, A)$, instead of the ordinary edge $(Y, -7, A)$. According to the analysis of Morris [53], the conditional constraint represented by this wait edge must be satisfied by any valid dynamic execution strategy for this network.

Definition 4 *(ESTNU graph).* An ESTNU graph is a tuple, $(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, where $(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u)$ is an STNU graph and $\mathcal{E}_w$ is a set of wait edges, each of the form $(V, C{:}{-}w, A)$, where A and C are the endpoints of some UC edge $(C, C{:}{-}y, A) \in \mathcal{E}_u$, $V \in \mathcal{T} \backslash \{C\}$, and $w \in \mathbb{R}$.

As will be seen, Morris [54] defined the dispatchability of an ESTNU in terms of its STN *projections.* A projection of an ESTNU is the STN that results from assigning fixed durations to its contingent links [56,54,37]. Each such assignment of fixed durations is represented by a *situation* [56].

Definition 5 *(Situation).* Let $\mathcal{S}$ be an STNU (or ESTNU) with k contingent links whose duration ranges are $[x_1, y_1]$, ..., $[x_k, y_k]$. A *situation* for $\mathcal{S}$ is a k-tuple $\omega = (\omega_1, \omega_2, \ldots, \omega_k)$ where $\omega_i \in [x_i, y_i]$ for each $i \in \{1, 2, \ldots, k\}$. If C is the contingent timepoint for a link (A, x, y, C), then the duration $C - A$ in the situation ω may be notated as ω_c.

A situation not only specifies the duration of each contingent link, it also determines the impact of each wait. This information is captured by the *projection* of an ESTNU [55,36].

Definition 6 *(Projection of an ESTNU).* For an ESTNU graph $\mathcal{G} = (\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, and a situation ω, the *projection* of $\mathcal{G}$ onto ω is the STN graph $\mathcal{G}_\omega = (\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_l^\omega \cup \mathcal{E}_u^\omega \cup \mathcal{E}_w^\omega)$, where:

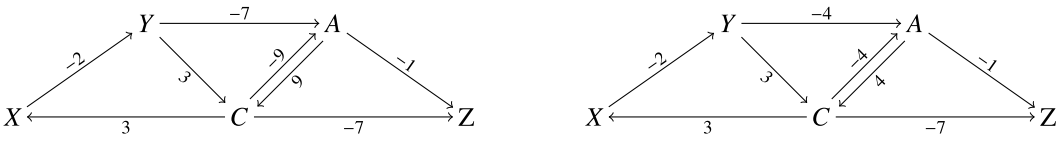
$$\begin{array}{lll} \mathcal{E}_l^\omega & = & \{(A_i, \omega_i, C_i) \quad | \exists (A_i, c_i{:}x_i, C_i) \in \mathcal{E}_l\} \\ \mathcal{E}_u^\omega & = & \{(C_i, -\omega_i, A_i) \ | \exists (C_i, C_i{:}-y_i, A_i) \in \mathcal{E}_u\} \\ \mathcal{E}_w^\omega & - & \{(V, \delta_i, A_i) \quad | \exists (V, C_i{:}-v, A_i) \in \mathcal{E}_w\}, \textit{where } \delta_i \text{ abbreviates } \max\{-\omega_i, -v\} \end{array}$$

Similarly, for any path $\mathcal{P}$ in $\mathcal{G}$, the *projection* of $\mathcal{P}$ in the situation ω, is the ordinary path $\mathcal{P}_\omega$ whose edges are obtained by projecting each edge in $\mathcal{P}$. In particular, each LC edge $(A, c{:}x, C)$ projects to (A, ω_c, C), each UC edge $(C, C{:}{-}y, A)$ projects to $(C, -\omega_c, A)$, and each wait edge $(V, C{:}{-}v, A)$ projects to (V, δ, A), where $\delta = \max\{-\omega_c, -v\}$.

The edges in $\mathcal{E}_l^\omega$ and $\mathcal{E}_u^\omega$ fix each duration $C_i - A_i$ to the value ω_i. The edges in $\mathcal{E}_w^\omega$ reflect the *effective* constraints imposed by the waits in $\mathcal{E}_w$ in the situation ω. For example, consider the wait edge $(Y, C{:}{-}7, A)$ from Fig. 10b. In the situation where $\omega_c = C - A = 4$, the wait projects onto $(Y, -4, A)$, reflecting that the conditional constraint vanishes when C executes early at $A + 4$. (Note that $\max\{-4, -7\} = -4$.) But if $\omega_c = 9$, then the wait projects onto $(Y, -7, A)$, reflecting that Y must wait the full 7 units after A, since C did not execute early. (Note that $\max\{-9, -7\} = -7$.) Fig. 11 shows the corresponding projections for the ESTNU from Fig. 10b.

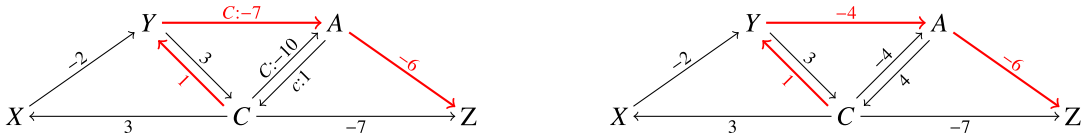
When first formalizing dispatchability for ESTNUs, Morris [54] argued that any valid dynamic execution strategy for an ESTNU, when restricted to a particular situation (unknown to the strategy), was equivalent to a dispatchable execution of the corresponding STN projection (equivalently, to a run of the RTE algorithm on that projection). Therefore, he *defined* the dispatchability of an ESTNU in terms of the dispatchability of its STN projections, as follows.

[11] Whereas the formal work on STN dispatchability first defined dispatchability as a performance guarantee with respect to the RTE algorithm and then proved that every dispatchable STN was vee-path complete, Morris' original approach to ESTNU dispatchability went in the opposite direction: defining an ESTNU to be dispatchable if each of its projections was STN-dispatchable, and then arguing that a dispatchable ESTNU would necessarily be successfully executed by a real-time execution algorithm. Hunsberger and Posenato's work confirmed that one could equivalently mirror the STN approach by first defining ESTNU dispatchability in terms of a performance guarantee with respect to the `RTE*` algorithm, and then proving that the projections of every dispatchable ESTNU would necessarily be STN-dispatchable.



(a) Projection where $\omega_c = C - A = 9$ (b) Projection where $\omega_c = C - A = 4$

Fig. 11. Two projections of the sample ESTNU from Fig. 10b.



(a) A dispatchable ESTNU (b) Its projection where $\omega_c = C - A = 4$

Fig. 12. A dispatchable ESTNU (left) equivalent to the STNU from Fig. 1b and the ESTNU from Fig. 10b; and one of its projections (right).

Definition 7 *(ESTNU dispatchability [54]).* An ESTNU is dispatchable if *all* of its projections are STN-dispatchable.

Since STN-dispatchability can be checked using the graphical characterization in terms of shortest vee-paths (cf. Theorem 1), this definition for ESTNU-dispatchability is attractive. For example, it is easy to check that the sample ESTNU graph in Fig. 1b is not ESTNU-dispatchable since both of its projections shown in Fig. 11 are not STN-dispatchable. For example, there is no SVP from C to Y in either projection; and there is no SVP from A to Z in the projection where $C - A = 4$. (Similar remarks apply to the sample STNU from Fig. 1b.) In contrast, Fig. 12a shows a *dispatchable* ESTNU that is equivalent to both the STNU from Fig. 10a (without the dotted edge) and the ESTNU from Fig. 1b.[12] Fig. 12b shows the projection for which $\omega_c = C - A = 4$, which *is* STN-dispatchable.

Morris [54] described a modification of his $O(n^3)$-time DC-checking algorithm for STNUs so that, when given a DC STNU as input, it generated an equivalent dispatchable ESTNU as output, thereby ensuring that each DC STNU has an equivalent dispatchable ESTNU. Hunsberger and Posenato [37] later presented an $O(mn + kn^2 + n^2 \log n)$-time algorithm, called $\texttt{FD}_{\text{STNU}}$, that is faster on sparse ESTNU graphs.

Later, Hunsberger and Posenato [41] confirmed the correctness of Morris' insights regarding ESTNU dispatchability by: (1) formally defining a real-time execution algorithm for ESTNUs, called RTE*, that preserves maximal flexibility during execution while requiring minimal real-time computation; and (2) proving that running their RTE* algorithm on any ESTNU that satisfied Morris' definition of dispatchability (i.e., Definition 7) would necessarily result in an execution sequence that satisfied all of the ESTNU's constraints, no matter how the contingent durations turned out. That result provides a solid theoretical foundation for ESTNU dispatchability.

The rest of this section provides more details on the RTE* algorithm and the central theoretical result. Section 7 presents a new algorithm for solving the MinDispESTNU problem (i.e., for computing equivalent dispatchable ESTNUs having a *minimal* number of edges), which is important for limiting the amount of local constraint propagation during execution.

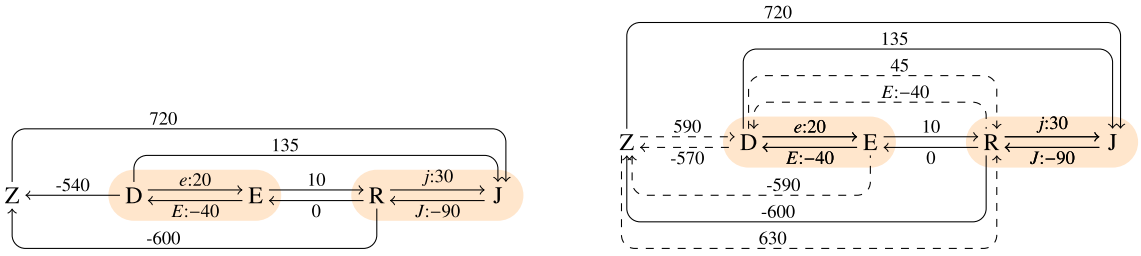
Enforcing dispatchability for the repair scenario. Fig. 13b shows the *dispatchable* ESTNU obtained by applying the $\texttt{FD}_{\text{STNU}}$ algorithm to the Repair-Scenario STNU from Fig. 1b (repeated in Fig. 13a). The dashed edges indicate the (ordinary and wait) constraints added by $\texttt{FD}_{\text{STNU}}$. The ESTNU contains all necessary constraints to ensure a successful incremental execution by the RTE* algorithm.

6.3. A real-time execution algorithm for ESTNUs

This section specifies a real-time execution algorithm for ESTNUs, called RTE*, whose high-level iterative operation is given as Algorithm 17 and whose helper functions are given as Algorithms 18–22. All of the pseudocode in this section is drawn from Hunsberger and Posenato [41], with some extra details concerning the efficient implementation of *activated waits*. The type of execution decisions and possible outcomes follow Hunsberger's characterization of dynamic execution strategies in terms of *real-time execution decisions* (RTEDs) [30].

On each iteration, the algorithm first generates (at Line 3) an execution decision, called Δ, which, in successful instances, has the form `wait` (i.e., wait for some contingent timepoint to execute) or (t, V) (i.e., if nothing happens before time t, execute the timepoint V at time t). The outcome of either type of decision depends on whether any contingent timepoints

[12] In this context, an STNU and ESTNU are called equivalent if they share the same set of valid dynamic execution strategies.



(a) The Repair-Scenario STNU from Figure 1b

(b) The dispatchable ESTNU obtained by running the FD_{STNU} algorithm (new edges dashed)

Fig. 13. Enforcing dispatchability for the Repair-Scenario STNU.

Algorithm 17: $\texttt{RTE}^*$: a real-time execution algorithm for ESTNUs.

Input: $\mathcal{G} = (\mathcal{T}_x \cup \mathcal{T}_c, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, an ESTNU graph
Output: A function (equivalently, a set of variable assignments), $f : \mathcal{T} \to \mathbb{R}$; or `fail`

1 `info` := $\texttt{RTE}^*_{init}$ $(\mathcal{G})$ *// Initialize an instance of an `RTEdata` data structure*
2 **while** $\texttt{info}.\mathcal{U}_x \cup \texttt{info}.\mathcal{U}_c \neq \emptyset$ **do** *// While some timepoints not yet executed*
3 Δ := $\texttt{RTE}^*_{genDecn}(\texttt{info})$ *// Generate next execution decision: `fail`; `wait`; or (t, V) for some $t \in \mathbb{R}$ and $V \subseteq \mathcal{T}_x$*
4 **if** $\Delta = \texttt{fail}$ **then return** `fail`
5 $(\rho, \tau) := \texttt{Observe}_c(\mathcal{G}, \texttt{info}, \Delta)$ *// Did any contingent timepoints (in ρ) happen to execute at some time $\tau \leq t$?*
6 `info` := $\texttt{RTE}^*_{update}(\texttt{info}, \Delta, (\rho, \tau))$ *// Update info in `info` to reflect outcome of the execution decision Δ, given (ρ, τ)*
7 **if** $\texttt{info} = \texttt{fail}$ **then return** `fail`
8 **return** $\texttt{info}.f$ *// Return the set of assignments to the timepoints in $\mathcal{T}$*

happen to execute, which is not controlled by the $\texttt{RTE}^*$ algorithm, but only *observed*. The observation step is represented here by an *oracle*, $\texttt{Observe}_c$ (at Line 5). The oracle non-deterministically outputs a pair (ρ, τ) representing that the contingent timepoints in the (possibly empty) set $\tau \subseteq \mathcal{U}_c$ happened to execute at time ρ. The only constraints on the oracle are that each contingent duration must fall within its specified bounds. Given the observation, the outcome of either type of decision is fully determined and is used to update the contents of the `info` data structure (at Line 6). In successful instances, the $\texttt{RTE}^*$ algorithm outputs (at Line 8) a complete set of variable assignments for the timepoints in $\mathcal{T}$ (equivalently, a function $f : \mathcal{T} \to \mathbb{R}$).

Algorithm 18: $\texttt{RTE}^*_{init}$: Initialize a new instance of an `RTEdata` data structure.

Input: $\mathcal{G} = (\mathcal{T}_x \cup \mathcal{T}_c, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, an ESTNU graph
Output: `info`, a suitably initialized `RTEdata` structure

1 `info` = new(`RTEdata`)
2 $\texttt{info}.\mathcal{U}_x := \mathcal{T}_x$; $\texttt{info}.\mathcal{U}_c := \mathcal{T}_c$; $\texttt{info.now} = 0$; $\texttt{info}.f = \emptyset$
3 **foreach** $X \in \mathcal{T}_x$ **do** $\texttt{numNegs}[X] := 0$ *// Initialize counts of negative ordinary or wait edges emanating from each $X \in \mathcal{T}_x$*
4 **foreach** $(U, \delta, V) \in \mathcal{E}_o$ *such that* $\delta < 0$ **do** $\texttt{numNegs}[U] := \texttt{numNegs}[U] + 1$
5 **foreach** $(U, C\!:\!-w, A) \in \mathcal{E}_w$ **do** $\texttt{numNegs}[U] := \texttt{numNegs}[U] + 1$
6 $\texttt{info.Enabs}_x = \{X \in \mathcal{T}_x \mid \texttt{numNegs}[X] = 0\}$ *// A hash-table of enabled timepoints*
7 **foreach** $X \in \mathcal{T}_x$ **do**
8 $\texttt{info.Win}(X)$:= $[0, \infty)$
9 $\texttt{info.ActWts}(X) := \emptyset$ *// For each X, $\texttt{ActWts}(X)$ is a max priority queue of activated waits for X*
10 **return** `info`

The $\texttt{RTE}^*$ algorithm keeps track of relevant information in an instance `info` of an `RTEdata` data structure that has the following fields: $\mathcal{U}_x$, the unexecuted *executable* timepoints; $\mathcal{U}_c$, the unexecuted *contingent* timepoints; `numNegs`, a vector where for each $X \in \mathcal{T}_x$, $\texttt{numNegs}[X]$ is the number of negative-length ordinary or wait edges from X to *as-yet-unexecuted* timepoints (see below); $\texttt{Enabs}_x$, a hash-table containing the *enabled* executable timepoints; `now`, the current time; f, a set of variable assignments; $\texttt{ActWts}(X)$, a vector where for each $X \in \mathcal{T}_x$, $\texttt{ActWts}(X)$ is a max priority queue of the *activated waits* for X (see below); and for each executable timepoint $X \in \mathcal{T}_x$, $\texttt{Win}(X) = [\texttt{lb}(X), \texttt{ub}(X)]$, the time window for X. The $\texttt{RTE}^*_{init}$ algorithm (Algorithm 18) handles the initialization of the `info` structure. Note that for ESTNUs, an *executable* timepoint X is enabled if all of its outgoing negative edges—*including wait edges*—point at already executed timepoints (i.e., if $\texttt{numNegs}[X]$ has hit 0).

Activated waits. A *wait* edge such as $(X, C{:}{-}w, A)$ represents a conditional constraint that *as long as C has not yet executed, X* must wait at least w after A. In what follows, a wait labeled by C is called a *C-wait.* Once the activation timepoint A for the contingent link (A, x, y, C) has been executed, say, at some time a, we say that the wait edge has been *activated*, which the RTE* algorithm keeps track of by inserting an entry $(a + w, C)$ into the max priority queue $\texttt{ActWts}(X)$. There are two ways for this wait to be satisfied: C can execute early (i.e., before time $a + w$) or the wait can expire (i.e., the current time passes $a + w$). In response to either event, the entry $(a + w, C)$ can be removed from $\texttt{ActWts}(X)$; however, for efficiency, this is done lazily, as described later on.

Generate execution decision. A real-time execution decision can have one of two forms: `wait` or (t, χ), where $t \in \mathbb{R}$ and $\chi \subseteq \mathcal{T}_x$. A `wait` decision can be glossed as "wait for a contingent timepoint to execute". A (t, χ) decision can be glossed as "if no contingent timepoints happen to execute before time t, then execute the timepoints in the set χ at time t". The RTE* algorithm assumes, like most work on STNUs, that an execution strategy can react instantaneously to execution events. As a result, it suffices to generate decisions where the set χ contains a single timepoint V.

Algorithm 19: $\texttt{RTE}^*_{genDecn}$: Generate the next execution decision.

Input: `info`, an instance of an `RTEdata` structure
Output: An execution decision: `wait` or (t, V), where $t \in \mathbb{R}$ and $V \subseteq \mathcal{T}_x$; or `fail`

1 **if** $\texttt{info.Enabs}_x = \emptyset$ **then return** `wait` // If no enabled timepoints, can only wait for a contingent TP to execute
2 **foreach** $X \in \texttt{info.Enabs}_x$ **do**
3 $\quad \texttt{maxWaitTime}(X) = \max\{a + w \mid \exists (a + w, C) \in \texttt{info.ActWts}(X) \text{ where } C \in \mathcal{U}_c\}$ // Max wait time for X; or $-\infty$
4 $\quad \texttt{glb}(X) = \max\{\texttt{info.lb}(X), \texttt{maxWaitTime}(X)\}$ // Effective greatest lower bound for X
5 $t_L = \min\{\texttt{glb}(X) \mid X \in \texttt{info.Enabs}_x\}$ // Earliest possible time that an enabled timepoint can be executed next
6 $t_U = \min\{\texttt{info.ub}(X) \mid X \in \texttt{info.Enabs}_x\}$ // Latest possible time by which some enabled timepoint must be executed
7 **if** $[t_L, t_U] \cap [\texttt{info.now}, \infty) = \emptyset$ **then return** `fail` // If maximum time window is closed, then fail
8 Select any $V \in \texttt{info.Enabs}_x$ for which $[\texttt{glb}(X), \texttt{ub}(X)] \cap [\texttt{info.now}, t_U] \neq \emptyset$ // Select a timepoint V to execute next
9 Select any $t \in [\texttt{glb}(V), \texttt{ub}(V)] \cap [\texttt{info.now}, t_U]$ // Select a time at which to execute V
10 **return** (t, V) // Return the decision to execute V at time t

$\texttt{RTE}^*_{genDecn}$ (Algorithm 19) computes the next RTED for one iteration of the RTE* algorithm. First (at Line 1), if there are no enabled timepoints, then the only viable decision is `wait`. Otherwise, the algorithm generates an RTED of the form (t, V) for some $t \in \mathbb{R}$ and some enabled timepoint V. Lines 2–4 compute, for each enabled timepoint X, the maximum wait time, $\texttt{maxWaitTime}(X)$, among all of X's activated waits (or $-\infty$ if there are none), and then compares that with the lower-bound $\texttt{lb}(X)$ from X's time window to generate the earliest time, $\texttt{glb}(X)$, at which X could be executed.[13] For efficiency, each $\texttt{ActWts}(X)$ set is implemented as a max priority queue, with entries of the form $(a + w, C)$, where $a + w$ is the lower bound corresponding to a wait edge $(X, C{:}{-}w, A)$, where A was executed at time a. In addition, obsolete entries corresponding to waits for which the contingent timepoint C has already been executed are removed lazily. In particular, when computing $\texttt{maxWaitTime}(X)$, the topmost entry of the queue is peeked at. If it represents a wait for which the contingent timepoint has not yet been executed, then its wait time, $a + w$, becomes the value of $\texttt{maxWaitTime}(X)$. Otherwise, that max entry is popped and then the peek/pop process continues recursively until one is found for which the contingent timepoint has not yet been executed; or the queue becomes empty. (If an activated wait is encountered for which $a + w \leq \texttt{info.now}$, then all of the remaining entries correspond to discharged waits and hence can be popped off the queue.) Since there are at most nk activated waits across the entire execution of the network, a total of at most nk pops of maximum entries will be required, taking at most $O(nk \log(nk))$ time.

$\texttt{RTE}^*_{genDecn}$ continues (at Line 5) computing the earliest possible time t_L that *any* enabled timepoint could be executed next; and (at Line 6) the latest time by which *some* enabled timepoint *must* be executed. The algorithm fails if the interval between the earliest possible time and the latest does not include times at or after now (Line 7). Otherwise, it selects V to be any one of the enabled timepoints whose time window includes times in $[\texttt{info.now}, t_U]$ (Line 8); and any time $t \in [\texttt{glb}(V), \texttt{ub}(V)] \cap [\texttt{info.now}, t_U]$ at which to execute it (Line 9). (Note the flexibility inherent in the selection of both V and t.) Finally, it outputs the RTED (t, V) (Line 10).

Observation. Once the RTE* algorithm generates an execution decision—either `wait` or (t, V)—it must wait to see what happens (i.e., whether some contingent timepoints happen to execute). Since the execution of contingent timepoints is not controlled by the RTE* algorithm, it is represented here by an *oracle,* called $\texttt{Observe}_c$. The oracle non-deterministically decides whether to execute any contingent timepoints and, if so, when. For any contingent link (A, x, y, C), if A has already been executed at some time a, then C could be executed at any time in the interval $[a + x, a + y]$. If the pending execution decision is `wait` but there are no activated contingent links, then the oracle outputs $(\infty, \emptyset)$, indicating that RTE* would have to wait forever. If the pending decision is (t, V) for some $t \in \mathbb{R}$, then, depending on the bounds for the activated

[13] $\texttt{info.lb}(X)$ and $\texttt{info.ub}(X)$ respectively denote the lower and upper bounds of X's time window, $\texttt{info.Win}(X)$.

contingent links, the oracle may choose to execute one or more contingent timepoints at some time $\rho \leq t$; or not. In either case, its output has the form (ρ, τ), where $\rho \leq t$ and $\tau \subseteq \mathcal{T}_x$. Hunsberger and Posenato [41] provided pseudocode for a simulation of the oracle that can aid empirical evaluations of RTE*.

Algorithm 20: $\texttt{RTE}^*_{update}$: update information in `info`.

Input: $\mathcal{G}$, an ESTNU graph; `info`, an `RTEdata` instance; Δ, an RTED; and (ρ, τ), an observation
Output: Updated `info` or `fail`

1 **if** $\rho = \infty$ **then return** `fail` // Case 0: Failure (waiting forever)
2 **if** $\Delta = \texttt{wait}$ ***or*** $(\Delta = (t, V)$ ***and*** $\rho < t)$ **then** **HCE**$(\mathcal{G}, \texttt{info}, \rho, \tau)$ // Case 1: Only contingent timepoints execute at $\rho < t$
3 **else**
4 **HNCE**$(\mathcal{G}, \texttt{info}, t, V)$ // Case 2: Executable timepoint V executes at t
5 **if** $\tau \neq \emptyset$ **then** **HCE**$(\mathcal{G}, \texttt{info}, t, \tau)$ // Case 3: Contingent timepoints ***also*** execute at t
6 $\texttt{info.now} := \rho$
7 **return** `info`

Update. Once the oracle decides whether any contingent timepoints shall execute, the outcome of the `wait` or (t, V) decision is fully determined. For a (t, V) decision, there are three possible outcomes: (1) some contingent timepoints executed at time $\rho < t$; (2) no contingent timepoints executed (i.e., $\tau = \emptyset$) so V is executed at time t; or (3) some contingent timepoints happened to execute *precisely* at time $\rho = t$, whence V is also executed at time t. (Case 3 is expected to be exceedingly rare in practice.) The **HCE** algorithm (handle contingent executions, Algorithm 21) handles the updates needed in response to contingent executions, which happen in Case 1 or Case 3 for a (t, V) decision or when $\rho < \infty$ for a `wait` decision. The **HNCE** algorithm (handle non-contingent executions, Algorithm 22) handles the updates needed when V executes at time t, which happens in Cases 1 and 2. For a `wait` decision, if there were no active contingent links, then $\rho = \infty$, meaning that the algorithm would have to wait forever, which is reported as a failure.

Algorithm 21: HCE: Handle contingent executions.

Input: $\mathcal{G}$, an ESTNU graph; `info`, an `RTEdata`; $\rho \in \mathbb{R}$, an execution time; $\tau \subseteq \mathcal{U}_c$, contingent timepoints executed at ρ
Result: `info` updated

1 **foreach** $C \in \tau$ **do**
2 $\texttt{info}.f(C) := \rho$
3 $\texttt{info}.\mathcal{U}_c := \texttt{info}.\mathcal{U}_c \backslash \{C\}$
4 **foreach** $(C, \delta_{cy}, Y) \in \mathcal{E}_o$ **do** $\texttt{info.ub}(Y) := \min\{\texttt{info.ub}(Y), \rho + \delta_{cy}\}$ // Propate UBs along edges leaving C
5 **foreach** $(W, \delta_{wc}, C) \in \mathcal{E}_o$ **do** // Propagate LBs along edges incoming to C
6 $\texttt{info.lb}(W) := \max\{\texttt{info.lb}(W), \rho - \delta_{wc}\}$
7 **if** $\delta_{wc} < 0$ **then** $\texttt{numNegs}[W] := \texttt{numNegs}[W] - 1$

The **HCE** algorithm (Algorithm 21) updates `info` in response to the observation of the contigent timepoints in τ executing at time ρ, as follows. For each $C \in \tau$, Lines 2–3 record that $\texttt{info}.f(C) = \rho$ and that C is no longer *unexecuted.* Lines 4–7 update the time windows for neighboring timepoints of C, exactly like in the RTE algorithm for STNs. Finally, since C is now unexecuted, all waits labeled by C are no longer needed; however, for efficiency, such waits are *not* removed from the relevant `ActWts` sets at this time. Instead, they are removed lazily by the $\texttt{RTE}^*_{genDecn}$ algorithm, as discussed previously. The **HCE** algorithm also (at Line 7) keeps track of the number of negative edges emanating from each timepoint, as discussed earlier.

Algorithm 22: HNCE: Handle a non-contingent execution.

Input: $\mathcal{G}$, an ESTNU; `info`, an `RTEdata` structure; $t \in \mathbb{R}$; $V \in \mathcal{U}_x$
Result: `info` updated

1 $\texttt{info}.f(V) := t$
2 $\texttt{info}.\mathcal{U}_x := \texttt{info}.\mathcal{U}_x \backslash \{V\}$
3 **foreach** $(V, \delta_{vy}, Y) \in \mathcal{E}_o$ **do** $\texttt{info.ub}(Y) := \min\{\texttt{info.ub}(Y), t + \delta_{vy}\}$ // Propate UBs along edges leaving V
4 **foreach** $(W, \delta_{wv}, V) \in \mathcal{E}_o$ **do** // Propagate LBs along edges incoming to V
5 $\texttt{info.lb}(W) := \max\{\texttt{info.lb}(W), t - \delta_{wv}\}$
6 **if** $\delta_{wv} < 0$ **then** $\texttt{numNegs}[W] := \texttt{numNegs}[W] - 1$
7 **if** *V is the activation timepoint for some contingent timepoint C* **then** // Record activated waits labeled by C
8 **foreach** $(Y, C{:}-w, V) \in \mathcal{E}_w$ **do** Insert $(t + w, C)$ into $\texttt{info.ActWts}(Y)$

Table 5
Sample runs of the RTE* algorithm on the *dispatchable* ESTNU from Fig. 12a.

Iter.	Win(A)	Win(X)	Win(Y)	ActWts(A)	ActWts(X)	ActWts(Y)	now	Enabs$_x$	Decn	Obs	Exec
Init.	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	{Z}	$(0,Z)$	$(0,\emptyset)$	$Z:=0$
1	$[6,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	$\{A\}$	$(7,A)$	$(7,\emptyset)$	$A:=7$
2	–	$[0,\infty)$	$[0,\infty)$	–	$\emptyset$	$\{(14,C)\}$	7	$\{Y\}$	$(16,Y)$	$(12,\{C\})$	$C:=12$
3	–	$[0,15]$	$[0,13]$	–	$\emptyset$	$\emptyset$	12	$\{Y\}$	$(13,Y)$	$(13,\emptyset)$	$Y:=13$
4	–	$[15,15]$	–	–	$\emptyset$	–	13	$\{X\}$	$(15,X)$	$(15,\emptyset)$	$X:=15$

(a) Sample run where C executes early (at $A+5$)

Iter.	Win(A)	Win(X)	Win(Y)	ActWts(A)	ActWts(X)	ActWts(Y)	now	Enabs$_x$	Decn	Obs	Exec
Init.	$[0,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	{Z}	$(0,Z)$	$(0,\emptyset)$	$Z:=0$
1	$[6,\infty)$	$[0,\infty)$	$[0,\infty)$	$\emptyset$	$\emptyset$	$\emptyset$	0	$\{A\}$	$(7,A)$	$(7,\emptyset)$	$A:=7$
2	–	$[0,\infty)$	$[0,\infty)$	–	$\emptyset$	$\{(14,C)\}$	7	$\{Y\}$	$(16,Y)$	$(16,\emptyset)$	$Y:=16$
3	–	$[18,\infty)$	–	–	$\emptyset$	–	16	$\{X\}$	$(22,X)$	$(17,\{C\})$	$C:=17$
4	–	$[18,20]$	–	–	$\emptyset$	–	17	$\{X\}$	$(19,X)$	$(19,\emptyset)$	$X:=19$

(b) Sample run where C executes late (at $A+10$)

The updates done by **HNCE** (Algorithm 22) in response to V executing at time t are similar to the updates done by **HCE** for each contingent timepoint C. However, in addition, if V happens to be the activation timepoint for some contingent timepoint C, then all waits labeled by C must be entered into the relevant ActWts queues (Lines 7–8).

Table 5 shows sample runs of the RTE* algorithm on the *dispatchable* ESTNU from Fig. 12a. In Table 5a, C executes early (at $A+5$); in Table 5b, C executes late (at $A+10$). Both runs result in variable assignments that satisfy all of the constraints in the network. In row 3 of Table 5a, ActWts(Y) is shown as empty. Lazy processing of ActWts(Y) would have left the entry $(16, C)$ in ActWts(Y) until later, when RTE$^*_{genDecn}$ needed to compute maxWaitTime(Y). At that point, peeking at the max entry in ActWts(Y) would have revealed that the wait labeled by C was discharged since C had already been executed (i.e., $C \notin \mathcal{U}_c$). That entry would then have been popped, clearing the queue.

RTE* *complexity.* The worst-case complexity of the RTE* algorithm is similar to that of the RTE algorithm except for the maintenance of the ActWts sets which, as described earlier, is $O(nk\log(nk))$. Therefore, the overall complexity of RTE* is $O(m + n\log n + nk\log(nk)) = O(m + nk\log(nk))$. Finally, from the perspective of the RTE* algorithm, the oracle presents observations in real time with no associated computational cost.

Dispatchability results. Hunsberger and Posenato [41] proved the following central results.

Theorem 2. *Let $\mathcal{G}$ be any ESTNU graph. Every run of the RTE* algorithm on $\mathcal{G}$ corresponds to a run of the RTE algorithm for STNs on some STN projection $\mathcal{G}_\omega$ of $\mathcal{G}$, yielding the same variable assignments to the timepoints in $\mathcal{T}$.*

Corollary 1. *An ESTNU $\mathcal{G}$ is dispatchable if and only if every run of the RTE* algorithm on $\mathcal{G}$ outputs a solution for the ordinary constraints in $\mathcal{G}$.*

Equivalently, we may *define* an ESTNU to be dispatchable if and only if every run of RTE* generates a solution and then take as a theorem the fact that an ESTNU is dispatchable if and only if every one of its STN projections is dispatchable (as an STN). The characterization of ESTNU dispatchability in terms of the dispatchability of its STN projections is invaluable in proving the correctness of algorithms related to ESTNU dispatchability since, together with Theorem 1, it connects ESTNU dispatchability to the existence of shortest vee-paths in STN projections.

7. Algorithms for solving the MinDispESTNU problem

Most DC-checking algorithms for STNUs do not guarantee a dispatchable output, but Morris [54] indicated that his $O(n^3)$-time DC-checking algorithm could be modified to do so. More recently, Hunsberger and Posenato [37] presented an $O(mn + kn^2 + n^2\log n)$-time algorithm for creating an equivalent dispatchable network, called FD$_{\text{STNU}}$, that is faster on sparse networks. Although these were the first algorithms to guarantee an equivalent dispatchable ESTNU as output, they do not provide any guarantees about the *number of edges* in their output. Since the number of edges directly impacts the computations done during execution (e.g., by the RTE* algorithm), more recent work has focused on solving the MinDispESTNU problem: computing an equivalent dispatchable ESTNU having a *minimal number of edges.*

Definition 8 *(MinDispESTNU Problem).* Given a DC STNU (or ESTNU) $\mathcal{G}$, the *MinDispESTNU problem* is the problem of computing an equivalent dispatchable ESTNU $\mathcal{G}'$ having a minimal number of edges. Such an ESTNU may be called a μESTNU for $\mathcal{G}$.

This section addresses the first algorithms for solving the MinDispESTNU problem. The first algorithm, called $\texttt{minDisp}_{\text{ESTNU}}$ [39], has a complexity of $O(kn^3)$ time, where n is the number of nodes and k the number of contingent links. The second algorithm, called $\texttt{minDisp}^{+}_{\text{ESTNU}}$, is a new algorithm whose complexity is $O(n^3 + k^2 n \log n)$ time. Since $\texttt{minDisp}^{+}_{\text{ESTNU}}$ is a modification of $\texttt{minDisp}_{\text{ESTNU}}$, the rest of this section first focuses on $\texttt{minDisp}_{\text{ESTNU}}$, before addressing the improvements in $\texttt{minDisp}^{+}_{\text{ESTNU}}$.

7.1. Preliminary observations

Suppose $\mathcal{G}$ is a dispatchable ESTNU and that $e = (V, \delta, W)$ is an ordinary edge in $\mathcal{G}$. If every projection of $\mathcal{G}$ contains a shortest vee-path (SVP) from V to W whose length is less than δ, then e can be removed from the network without threatening its dispatchability. However, as will be seen, this determination can be complicated by the fact that the SVPs from V to W may follow different routes in different projections. The algorithms presented in this section address this issue by exploring certain structures in the ESTNU graph called *diamond* structures. In addition, they also address how to determine which wait edges can be removed without threatening the network's dispatchability.

Definition 9 *(Path lengths in different projections).* Let $\mathcal{G}$ be any ESTNU graph; $\mathcal{P}$ any path in $\mathcal{G}$; and ω any situation. If $\mathcal{P}$ has only ordinary edges, then $|\mathcal{P}|$ denotes its length; and $\mathrm{d}(V, W)$ denotes the length of the shortest path from V to W that has only *ordinary* edges. More generally, since the projection of *any* path $\mathcal{P}$ is, by definition, an ordinary path, its length is denoted by $|\mathcal{P}_\omega|$. We may also refer to $|\mathcal{P}_\omega|$ as the length of $\mathcal{P}$ in the situation ω and notate it as $|\mathcal{P}|_\omega = |\mathcal{P}_\omega|$. For any V and W, $\mathrm{d}_\omega(V, W)$ denotes the length of any SVP from V to W in $\mathcal{G}_\omega$; and $d_*(V, W) = \max_\omega\{\mathrm{d}_\omega(V, W)\}$ denotes the maximum such length *across all projections.* If context allows, we may notate a path by listing its timepoints (e.g., noting a path from V to A to C as VAC, and the length of its projection as $|VAC|_\omega$).

Using this notation, if $e = (V, \delta, W)$ is an ordinary edge in a dispatchable ESTNU $\mathcal{G}$, then e can be removed from $\mathcal{G}$ without threatening its dispatchability if $d_*(V, W) < \delta$. In addition, if $d_*(V, W) = \delta$, then e can be safely removed if every projection $\mathcal{G}_\omega$ has an SVP from V to W that does not use e.

7.2. The $\texttt{minDisp}_{\text{ESTNU}}$ algorithm

This section summarizes $\texttt{minDisp}_{\text{ESTNU}}$, the first algorithm for solving the MinDispESTNU problem, due to Hunsberger and Posenato [39]. Appendix A presents a rigorous and in-depth proof of the algorithm's correctness that greatly expands upon the previously published proof sketch.

One of the main goals of the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm is to compute, for each pair of timepoints U and W, the value of $d_*(U, W)$, which can be understood not only as the maximum length of any SVP from U to W across all projections, but also as specifying the strongest ordinary constraint on $W - U$ that must be satisfied by any dynamic execution strategy. The basic approach of $\texttt{minDisp}_{\text{ESTNU}}$ is to generate ordinary "stand-in" edges that are entailed by various combinations of ESTNU edges, especially those entailed by "nested diamond structures". After inserting those stand-in edges into the network, $\texttt{minDisp}_{\text{ESTNU}}$ then uses an STN-dispatchability algorithm on the network's ordinary edges (including the stand-in edges) to determine which ordinary edges can be removed without threatening the ESTNU's dispatchability. It then separately determines which wait edges can be removed.

The $\texttt{minDisp}_{\text{ESTNU}}$ algorithm takes the following steps: (1) it fixes *weak* and *misleading* wait edges; (2) it generates ordinary *stand-in* edges that are entailed by individual labeled edges or possibly-nested diamond structures; (3) it runs an STN-dispatchability algorithm on the resulting set of ordinary edges; and (4) it removes wait edges that are not needed for ESTNU-dispatchability. Each of these steps is discussed in detail, below.

Fixing "weak" and "misleading" wait edges. In Fig. 14a, $(Y, C{:}{-}2, A)$, shown in red, is a *weak wait*, since the minimum duration of the contingent link $(A, 3, 10, C)$ is 3, which implies that C cannot execute before the wait time of 2 expires. So this wait is effectively unconditional and hence can be *replaced* by the ordinary edge $(Y, -2, A)$, shown in green. In general, the *Unconditional Unordered Reduction* rule of Morris et al. [56] ensures that any wait edge $(Y, C{:}{-}v, A)$ where (A, x, y, C) is the relevant contingent link can be replaced by the ordinary edge $(Y, -v, A)$ if $v \leq x$ (i.e., the wait time is less than the minimum duration of the contingent link). In contrast, $(W, C{:}{-}15, A)$, also shown in red, is a *misleading wait* since C *must* execute no later than 10 after A. This wait is fixed by changing the wait time to the maximum of 10: $(W, C{:}{-}10, A)$, shown in green. In general, any wait edge $(W, C{:}{-}w, A)$ where (A, x, y, C) is the relevant contingent link can be replaced by the wait edge $(W, C{:}{-}y, A)$ if $w > y$ (i.e., the wait time is greater than the maximum duration of the contingent link).

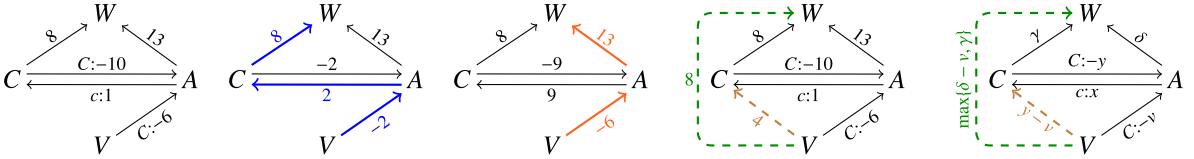
Definition 10 *(Regular wait edge).* A wait edge $(W, C{:}{-}v, A)$ associated with a contingent link (A, x, y, C) is called *regular* if it is neither weak nor misleading (i.e., if its wait amount v satisfies $x < v \leq y$).

7.2.1. Stand-in edges

After fixing all weak and misleading waits, the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm focuses on generating *stand-in* edges. Each stand-in edge is an *ordinary* edge that is used to make explicit a constraint entailed by individual labeled edges or com-

(a) Fixing *weak* and *misleading* waits (b) Initial stand-in edges

Fig. 14. Fixing wait edges and generating initial stand-in edges (dashed).



(a) ESTNU (b) $\omega_c = 2$ (c) $\omega_c = 9$ (d) Dashed stand-in edges (e) General *VACW* diamond

Fig. 15. A sample ESTNU, two of its projections with (colored) shortest vee-paths, and two (dashed) stand-in edges.

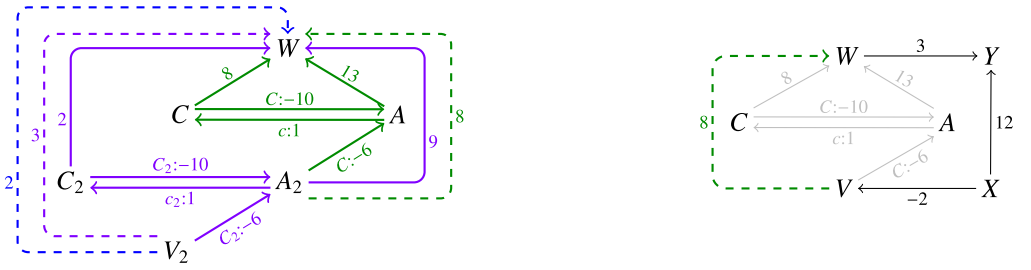
binations of ESTNU edges across different STN projections. The stand-in edges are subsequently used to determine what other edges can be removed from the network without threatening its dispatchability. Afterward, the stand-in edges are eventually removed from the network—because they are, by definition, entailed by other paths in the network.

Initial stand-in edges. Fig. 14b shows (dashed) stand-in edges that represent the strongest ordinary constraints that are entailed by various labeled ESTNU edges associated with the contingent link $(A, 3, 10, C)$. The stand-in edge for the LC edge $(A, c{:}3, C)$ is $(A, 10, C)$, representing that $C - A \leq 10$ in every projection. The stand-in edge for the UC edge $(A, C{:}{-}10, C)$ is $(C, -3, A)$, representing that $A - C \leq -3$ (i.e., $C - A \geq 3$) in every projection. For the wait edge $(V, C{:}{-}6, A)$, the stand-in edge is $(V, -3, A)$, representing that V must *unconditionally* wait at least 3 after A, since C cannot execute sooner than that. In general, the *unconditional Unordered Reduction* rule of Morris et al. [56] states that a (regular) wait edge $(V, C{:}{-}v, A)$ associated with a contingent link (A, x, y, C) entails an ordinary edge $(V, -x, A)$. In other words, if the conditional wait time v is greater than the minimum duration x, then V must *unconditionally* wait at least x after A. These observations motivate the following definition.

Definition 11 *(Stand-in edges for individual labeled edges).* Let (A, x, y, C) be any contingent link. The *stand-in* edge for the LC edge $(A, c{:}x, C)$ is the ordinary edge (A, y, C); the stand-in edge for the UC edge $(C, C{:}{-}y, A)$ is the ordinary edge $(C, -x, A)$; and the stand-in edge for the (regular) wait edge $(W, C{:}{-}v, A)$ is the ordinary edge $(W, -x, A)$.

The VAC rule. In Fig. 14b, there is one more (dashed) stand-in edge, $(V, 4, C)$, shown in brown. It is entailed by the two-edge path comprising the wait edge followed by the LC edge. In particular, in each projection, $|VAC|_\omega = \max\{-6, -\omega_c\} + \omega_c = \max\{\omega_c - 6, 0\} \leq 4$, since $\omega_c \leq 10$. In general, for any regular wait edge $(V, C{:}{-}v, A)$ and associated contingent link (A, x, y, C), the *VAC rule* generates the ordinary edge $(V, y - v, C)$.

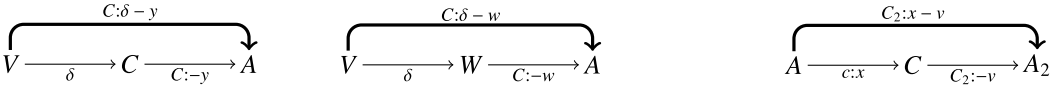
Stand-in edges for diamond structures. Although the kinds of stand-in edges discussed above are needed by the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm, stand-in edges can also be entailed by other structured combinations of edges, which we call *diamond* structures. For example, consider the ESTNU in Fig. 15a. Since it has only one contingent link, each projection ω is determined by the value ω_c that is assigned to $C - A$. In the projection where $\omega_c = 2$, shown in Fig. 15b, the SVP from V to W, indicated by thick blue edges, has length 8, whereas in the projection shown in Fig. 15c, where $\omega_c = 9$, the SVP from V to W, indicated by thick orange edges, has length 7. It is not hard to check that for this ESTNU, the length of the SVP from V to W is at most 8 in every projection. In particular, in projections where $\omega_c \leq 5$, $|VACW|_\omega = \max\{-\omega_c, -6\} + \omega_c + 8 = \max\{8, \omega_c + 2\} \leq 8$; but in projections where $\omega_c \geq 5$, $|VAW|_\omega = \max\{-\omega_c, -6\} + 13 = \max\{13 - \omega_c, 7\} \leq 8$. Hence, this combination of ordinary and labeled edges entails the stand-in edge $(V, 8, W)$, shown as green and dashed in Fig. 15d. For reference, the (brown and dashed) stand-in edge $(V, 4, C)$ generated by the VAC rule, as described earlier, is also shown in Fig. 15d. Fig. 15e shows the general case of deriving a (green and dashed) stand-in edge for a *VACW* diamond, as well as the (brown and dashed) stand-in edge generated by the VAC rule. As shown in the formal analysis in Appendix A, the length of the stand-in edge is $\max\{\delta - v, \gamma\}$, where $\delta - v$ is the length of the path *VAW* in the projection $\omega_c = y$; and γ is the length of *VCW* in the projection $\omega_c = x$. The value of $\theta = \delta - \gamma$ plays a key role. In particular, the length of the stand-in edge is γ if $\theta \in (x, v]$, but is $\delta - v$ if $\theta \in [v, y]$. If $\theta \notin (x, y]$, then the stand-in edge is not needed.



(a) Stand-in edges entailed by nested diamonds

(b) Using stand-in edge to dominate $(X, 12, Y)$

Fig. 16. Stand-in edges entailed by nested diamonds (left); using a stand-in edge to dominate the edge $(X, 12, Y)$ (right).



(a) Upper-Case Rule applied to a UC edge (left) or a wait edge (right)

(b) Cross-Case Rule

Fig. 17. Rules for generating wait edges, from Morris and Muscettola [58] (generated edges in bold).

Stand-in edges for nested diamonds. Fig. 16a shows a more complicated ESTNU, where the diamond structure involving the solid green edges is nested inside the diamond structure involving the solid purple edges. Ignoring the green edges, for now, the solid purple edges can be shown to entail the (purple, dashed) stand-in edge $(V_2, 3, W)$. In particular, in projections where $\omega_2 = C_2 - A_2 \leq 7$, the length of the path $V_2A_2C_2W$ is: $\max\{-\omega_2, -6\} + \omega_2 + 2 = \max\{2, \omega_2 - 4\} \leq 3$. In contrast, if $\omega_2 \geq 7$, the length of the alternative path V_2A_2W is: $\max\{-\omega_2, -6\} + 9 = \max\{9 - \omega_2, 3\} \leq 3$.

Next, since the green diamond is isomorphic to the diamond from Fig. 15a, it entails the (green, dashed) stand-in edge $(A_2, 8, W)$. But now, using that stand-in edge instead of the purple edge $(A_2, 9, W)$, a new analysis of the purple structure shows that it now entails a stronger (blue, dashed) stand-in edge $(V_2, 2, W)$. In other words, nested diamond structures can sometimes combine to entail stronger stand-in edges. Not only that, the order in which nested diamonds are analyzed can affect the overall computational effort required. For example, if the green diamond is analyzed first, then the purple diamond only needs to be analyzed once.

The importance of stand-in edges. The network in Fig. 16b includes the diamond structure from Fig. 15a (with its original ESTNU edges drawn in light gray), its (green, dashed) stand-in edge $(V, 8, W)$, and some additional edges. Assuming that the stand-in edge is present in the network, then the *vee-path XVWY*, which contains only ordinary edges, has length $-2 + 8 + 3 = 9$, which is less than the length of the edge $(X, 12, Y)$. That implies that the edge $(X, 12, Y)$ can be removed from the network without affecting its dispatchability. Had the pre-existing edge been $(X, 9, Y)$, it could still be removed, given the alternative vee-path. Crucially, this conclusion depended solely on ordinary edges and, therefore, in general, can be determined by applying an STN-dispatchability algorithm to those ordinary edges. (That is Step 3 of the `minDisp`$_{\text{ESTNU}}$ algorithm.) Returning to Fig. 16b, once the edge $(X, 12, Y)$ has been removed, the stand-in edge has played its role and can be discarded because, in each projection, either *XVAWY* or *XVACWY* will provide an SVP from X to Y.

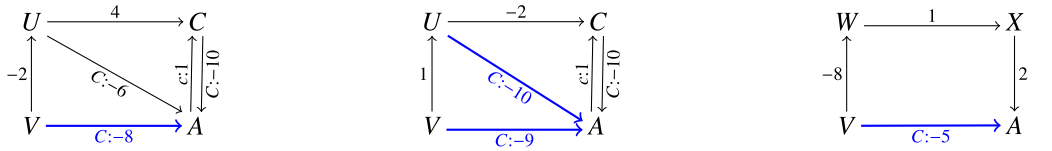
7.2.2. Removing unneeded wait edges

Computing a μESTNU also requires determining which wait edges can be removed without threatening the dispatchability of the network. There are three cases: wait edges that are dominated (1) by other wait edges, (2) by UC edges; or (3) by ordinary paths.

First, we review the rules of Morris and Muscettola [58] that are used to generate wait edges. Since these rules are sound, the waits they generate *must* be satisfied by *every* valid execution strategy.

The *Upper Case* (UC) rule in Fig. 17a has two cases. First, as shown on the left, it can be applied to an ordinary edge (V, δ, C) and the UC edge $(C, C{:}-y, A)$ to generate a (thick) wait edge $(V, C{:}\delta - y, A)$. Intuitively, since $C - A$ might be as big as y, if we want to satisfy the constraint $C - V \leq \delta$, then V must *wait* $(y - \delta)$ after A—unless C happens to execute early. Second, as shown on the right, the UC rule can also be applied to an ordinary edge (V, δ, W) and a wait edge $(W, C{:}-w, A)$ to generate a (thick) wait edge $(V, C{:}\delta - w, A)$. Intuitively, since W might have to wait w after A (unless C executes early), then to satisfy $W - V \leq \delta$, V must wait $w - \delta$ after A (unless C executes early).[14]

[14] Morris and Muscettola [58] refer to both the UC edge and all wait edges as upper-case edges, despite their very different semantics.



(a) Dominated by another wait (b) Dominated by a UC edge (c) Dominated by an ordinary path

Fig. 18. Examples of (blue) wait edges that can be removed without threatening ESTNU dispatcability.

The *Cross Case* (CC) rule, shown in Fig. 17b, combines the LC edge $(A, c{:}x, C)$ for one contingent link with a wait edge $(C, C_2{:}{-}v, A_2)$ labeled by a *different* contingent timepoint C_2 to generate a (thick) wait edge $(A, C_2{:}x - v, A_2)$. Intuitively, since C might be as little as x after A, but C must wait at least v after A_2 (unless C_2 executes early), then A must *wait* at least $(v - x)$ after A_2 (unless C_2 executes early).

Waits dominated by other waits. Fig. 18a shows an ESTNU having a (blue) wait edge $(V, C{:}{-}8, A)$ that is dominated by another wait edge $(U, C{:}{-}6, A)$. In particular, in any situation $\omega_c \in [1, 10]$, $|VUA|_{\omega_c} = -2 + \max\{-6, -\omega_c\} = \max\{-8, -2 - \omega_c\} \leq \max\{-8, -\omega_c\} = |(V, C{:}{-}8, A)|_{\omega_c}$. The key features behind this derivation are that the path from V to U (in this case, a single edge) has negative length (i.e., $\mathrm{d}(V, U) < 0$) and $\mathrm{d}(V, U) - u \leq -v$. In general, $|VUA|_{\omega_c} = \mathrm{d}(V, U) + \max\{-u, -\omega_c\} = \max\{\mathrm{d}(V, U) - u, \mathrm{d}(V, U) - \omega_c\} \leq \max\{-v, -\omega_c\} = |(V, C{:}{-}v, A)|_{\omega_c}$.

Waits dominated by UC edges. Fig. 18b shows two (blue) wait edges, $(U, C{:}{-}10, A)$ and $(V, C{:}{-}9, A)$, that are dominated by the UC edge $(C, C{:}{-}10, A)$ via the paths, *UCA* and *VUCA*, respectively.[15] For each $\omega_c \in [1, 10]$, $|UCA|_{\omega_c} = -2 + |(C{:}-10, A)|_{\omega_c} = -2 - \omega_c < -\omega_c \leq \max\{-\omega_c, -10\} = |(U, C{:}-10, A)|_{\omega_c}$. The key feature behind this derivation is that the path from U to C (in this case, a single edge) has negative length (i.e., $\mathrm{d}(U, C) = -2 < 0$). It can similarly be shown that $|VUCA|_{\omega_c} \leq |(V, C{:}-9, A)|_{\omega_c}$ for each $\omega_c \in [1, 10]$, again, the key feature being that $\mathrm{d}(V, C) < 0$.

Waits dominated by ordinary paths. In Fig. 18c, the (blue) wait $(V, C{:}-5, A)$ is dominated by the ordinary path *VWXA*, since for each ω_c, $|VWXA|_{\omega_c} = -5 \leq \max\{-5, -\omega_c\} = |(V, C{:}{-}5, A)|_{\omega_c}$.

These observations motivate the following definition.

Definition 12 *(Dominated wait edge).* Let $E = (V, C{:}{-}v, A)$ be any regular wait edge, where (A, x, y, C) is the relevant contingent link. E is *dominated* by an ordinary path $\mathcal{P}$ from V to A if $|\mathcal{P}| \leq -v$. E is *dominated* by the UC edge $(C, C{:}{-}y, A)$ if $d(V, C) < 0$. E is *dominated* by another wait edge $(U, C{:}{-}u, A)$ if $d(V, U) < 0$ and $d(V, U) - u \leq -v$.

7.2.3. Pseudocode for the $\texttt{minDisp}_{\mathrm{ESTNU}}$ algorithm

This section presents detailed pseudocode for the $\texttt{minDisp}_{\mathrm{ESTNU}}$ algorithm (cf. Algorithms 23–27). It takes a dispatchable ESTNU $\mathcal{G} = (\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$ as input, and generates as output a μESTNU (i.e., an equivalent dispatchable ESTNU with a minimal number of edges). Since the input is dispatchable, the algorithm must only determine which edges can be removed while preserving dispatchability.[16]

Algorithm 23: $\texttt{minDisp}_{\mathrm{ESTNU}}$: Solving the MinDispESTNU problem.

Input: $\mathcal{G} = (\mathcal{T},\ \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, a dispatchable ESTNU
Output: A μESTNU for $\mathcal{G}$

1 $(\mathcal{E}_o^{\mathrm{si}}, d) := \texttt{genStandIns}(\mathcal{T},\ \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$ // Compute the set of (ordinary) stand-in edges
2 $(\mathcal{T}, \mathcal{E}_o^*, \hat{\mathcal{E}}_{\mathrm{l}}, \hat{\mathcal{E}}_{\mathrm{u}}, \hat{\mathcal{E}}_{\mathrm{w}}) := \texttt{disp}_{\mathrm{STN}}(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_o^{\mathrm{si}}, \mathcal{E}_l, \mathcal{E}_u, \mathcal{E}_w)$ // STN dispatchability on ordinary edges, reorienting labeled edges
3 $\hat{\mathcal{E}}_o^* := \mathcal{E}_o^* \backslash \mathcal{E}_o^{\mathrm{si}}$ // Remove any remaining stand-in edges from $\mathcal{E}_o^*$
4 $\hat{\mathcal{E}}_{\mathrm{w}} := \hat{\mathcal{E}}_{\mathrm{w}} \backslash\ \texttt{markWaits}(\mathcal{T}_c, \hat{\mathcal{E}}_{\mathrm{w}}, d)$ // Remove dominated waits
5 **return** $\mathcal{G} = (\mathcal{T}, \hat{\mathcal{E}}_o^* \cup \hat{\mathcal{E}}_{\mathrm{l}} \cup \hat{\mathcal{E}}_{\mathrm{u}} \cup \hat{\mathcal{E}}_{\mathrm{w}})$

The high-level pseudocode (Algorithm 23) has four steps. First, it computes $\mathcal{E}_o^{\mathrm{si}}$, the set of (ordinary) stand-in edges entailed by both individual labeled edges and (possibly nested) diamond structures. These stand-in edges only play a supporting role in determining which other edges can be safely removed from the network. Since the stand-in edges are, by

[15] The Upper-Case rule from Fig. 17a would generate the *misleading* wait edges, $(U, C{:}{-}12, A)$ and $(V, C{:}{-}11, A)$. Fixing the first wait would yield $(U, C{:}{-}10, A)$, since 10 is the maximum duration of the contingent link. Afterward, the Upper-Case rule could be used to generate the wait edge $(V, C{:}{-}9, A)$. However, even if the wait edges are not fixed, a similar analysis shows that they would be still be dominated by the UC edge.

[16] For any given STNU, the $\mathrm{FD}_{\mathrm{STNU}}$ algorithm [37] can be used to generate an initial equivalent dispatchable STNU; or $\perp$ if the STNU is not DC.

definition, entailed by other edges, they will also be removed before the end of the algorithm. Second, the STN-based dispatchability algorithm from Tsamardinos et al. [72], called $\texttt{disp}_{\text{STN}}$ here, is used to transform the STN $(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_o^{\text{si}})$ into an equivalent dispatchable STN $(\mathcal{T}, \mathcal{E}_o^*)$ with a minimal number of edges (for the STN). Third, if any stand-in edges remain in $\mathcal{E}_o^*$, they are removed. Fourth, all wait edges that are dominated by (1) ordinary paths; (2) UC edges; or (3) other waits are removed from the network. Finally, $\texttt{minDisp}_{\text{ESTNU}}$ outputs the μESTNU. Each step of the algorithm is discussed in more detail below.

Generating stand-in edges. The stand-in edges are computed by the `genStandIns` algorithm (Algorithm 24) and stored in the set $\mathcal{E}_o^{\text{si}}$. Together, the edges in $\mathcal{E}_o \cup \mathcal{E}_o^{\text{si}}$ encode the strongest *ordinary* constraints entailed by the input ESTNU. `genStandIns` also computes the all-pairs shortest-paths (APSP) distance function d for the edges in $\mathcal{E}_o \cup \mathcal{E}_o^{\text{si}}$.

Algorithm 24: `genStandIns`: Generate all stand-in edges.

Input: $\mathcal{G} = (\mathcal{T}_x \cup \mathcal{T}_c,\ \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, a dispatchable ESTNU
Output: $(\mathcal{E}_o^{\text{si}}, d)$, where $\mathcal{E}_o^{\text{si}}$ is a set of (ordinary) stand-in edges, and d is the all-pairs shortest-paths function for $\mathcal{E}_o \cup \mathcal{E}_o^{\text{si}}$.

```
1 L := the contingent links associated with G
2 E_o^si := getInitStandins (G)          // Fix weak or misleading waits; stand-in edges for individual labeled edges and VAC rule
3 for i := 1 to k do                     // k = max depth of nested diamond structures
4     d := Johnson(T, E_o ∪ E_o^si)      // All-pairs shortest-paths (APSP) distances for ordinary paths
5     (E_o^si, dChange?) := getDiamondStandins (T, L, E_w, E_o^si, d)
6     if dChange? == ⊥ then return (E_o^si, d)   // Exit from the for loop
7 d := Johnson(T, E_o ∪ E_o^si)          // Must have done k iterations, so need to update d one last time
8 return (E_o^si, d)                     // At this point, d = d_*
```

First, at Line 2, `genStandIns` calls `getInitStandins` (Algorithm 25) to fix all weak or misleading wait edges, and compute the stand-in edges arising from individual labeled edges or applications of the VAC rule. In particular, Line 3 (of

Algorithm 25: `getInitStandins`: Generate stand-in edges entailed by individual labeled edges.

Input: $\mathcal{G} = (\mathcal{T}_x \cup \mathcal{T}_c,\ \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, a dispatchable ESTNU
Output: The set $\mathcal{E}_o^{\text{si}}$ of ordinary stand-in edges for the individual labeled edges in $\mathcal{G}$
Side Effect: Modifies $\mathcal{G}$ by fixing any weak or misleading wait edges

```
1 E_o^si := ∅
2 foreach (A, x, y, C) ∈ L do                                   // Collect stand-in edges for LC, UC and wait edges
3     E_o^si := E_o^si ∪ {(A, y, C), (C, −x, A)}                // Collect stand-in edges for LC and UC edges
4     foreach (V, C:−v, A) ∈ E_w do
5         if −v ≥ −x then E_w := E_w \ {(V, C:−v, A)}; E_o := E_o ∪ {(V, −v, A)}   // Replace weak wait edge by an ordinary edge
6         else
7             if −v < −y then E_w := E_w \ {(V, C:−v, A)} ∪ {V, C:−y, A)}          // Fix misleading wait by adjusting its wait time
8             E_o^si := E_o^si ∪ {(V, −x, A), (V, max{y − v, 0}, C)}               // Add stand-in edges for wait edge and from the VAC rule
9 return E_o^si
```

Algorithm 25) collects the stand-in edges for the LC and UC edges; Line 5 replaces any *weak* waits by ordinary edges; Line 7 adjusts the wait times of any *misleading* waits; and Line 8 collects the stand-in edges entailed by wait edges and applications of the VAC rule where, for a fixed misleading wait edge, the VAC rule generates the ordinary edge $(V, 0, C)$.

Next, the main loop of `genStandIns` (Lines 3–6) does at most k iterations, aiming to generate the stand-in edges entailed by all *VACW* "diamond" structures (recall the dashed green stand-in edge from Fig. 15d). Each iteration is handled by a call to the `getDiamondStandins` algorithm (Algorithm 26), whose Lines 5–11 check whether each *VACW* diamond structure yields a stand-in edge as illustrated in Fig. 15e. Since `getDiamondStandins` requires an updated distance function d, each iteration begins by calling Johnson's algorithm. The correctness of `genStandIns` and `getDiamondStandins` is proven in Appendix A.

Using the STN-dispatchability algorithm. Line 2 of $\texttt{minDisp}_{\text{ESTNU}}$ applies the *STN* dispatchability algorithm from Tsamardinos et al. [72], called $\texttt{disp}_{\text{STN}}$ here, to the STN $(\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_o^{\text{si}})$ to compute an equivalent, dispatchable STN $(\mathcal{T}, \mathcal{E}_o^*)$, typically by inserting some new ordinary edges and deleting others. The STN output by that algorithm is guaranteed to have the minimum number of edges needed for the STN's dispatchability. To achieve its $O(mn + n \log n)$ worst-case time, the $\texttt{disp}_{\text{STN}}$ algorithm identifies any *rigid components* in the input STN and then *reorients* its edges to make them incident to only the

Algorithm 26: `getDiamondStandins`: Generate stand-in edges entailed by diamond structures.

Input: $\mathcal{T}$, timepoints; $\mathcal{L}$, contingent links; $\mathcal{E}_w$, wait edges; $\mathcal{E}_o^{\text{si}}$, stand-in edges, d, distance function
Output: $(\mathcal{E}_o^{\text{si}}, dChange?)$, where $dChange? = \top$ if edges added to $\mathcal{E}_o^{\text{si}}$ require updating d

```
1  dChange? := ⊥
2  foreach (A, x, y, C) ∈ L do                                   // Explore VACW "diamond" structures (cf. Fig. 15d)
3      foreach (V, C:−v, A) ∈ E_w do
4          foreach W ∈ T \ {A, C, V} do
5              γ := d(C, W);   δ := d(A, W)
6              if γ < ∞ and δ < ∞ then
7                  if x < δ − γ ≤ y then                          // Applicability condition for stand-in edge (cf. Appendix A)
8                      θ := max{δ − v, γ}                         // θ = max{δ − v, γ} is the weight of the stand-in edge
9                      if θ ≤ d(V, W) then
10                         E_o^si := E_o^si ∪ {(V, θ, W)}         // The VACW diamond structure generates a stand-in edge!
11                         if θ < d(V, W) then dChange? := ⊤
12 return (E_o^si, dChange?)
```

representative timepoints of those rigid components.[17] If this reorientation of ordinary edges affects any stand-in edges associated with individual labeled edges (e.g., the kinds of edges shown as dashed in Fig. 14b), then our version of $\texttt{disp}_{\text{STN}}$ similarly reorients the corresponding labeled edges from the ESTNU. That is why $\texttt{disp}_{\text{STN}}$ takes the labeled edge sets, $\mathcal{E}_l, \mathcal{E}_u$ and $\mathcal{E}_w$, as extra inputs and returns the corresponding sets of reoriented edges, $\hat{\mathcal{E}}_{\text{l}}, \hat{\mathcal{E}}_{\text{u}}$ and $\hat{\mathcal{E}}_{\text{w}}$, as extra outputs. (The labeled edges are only reoriented; none are lost or added in the process.) After applying $\texttt{disp}_{\text{STN}}$, the stand-in edges in $\mathcal{E}_o^{\text{si}}$ have served their purpose and, therefore, any stand-in edges happening to remain in $\mathcal{E}_o^*$ are removed by $\texttt{minDisp}_{\text{ESTNU}}$ (Algorithm 23, Line 3).

Algorithm 27: `markWaits`: Mark wait edges for removal.

Input: $\mathcal{T}_c$, the contingent TPs; $\hat{\mathcal{E}}_{\text{w}}$, the wait edges; d, a distance function
Output: A set $\mathcal{E}_{\text{w}}^m \subseteq \hat{\mathcal{E}}_{\text{w}}$ of wait edges marked for removal

```
1  E_w^m := ∅
2  foreach (V, C:−v, A) ∈ Ê_w do                                 // Collect waits dominated by ordinary paths, UC edges, or other waits
3      if d(V, A) ≤ −v or d(V, C) < 0 then E_w^m := E_w^m ∪ {(V, C:−v, A)}   // Dominated by an ordinary path or the UC edge
4      else
5          foreach U ∈ T | ∃(U, C:−u, A) ∈ Ê_w do
6              if d(V, U) < 0 and d(V, U) − u ≤ −v then E_w^m := E_w^m ∪ {(V, C:−v, A)}   // Dominated by another wait
7  return E_w^m
```

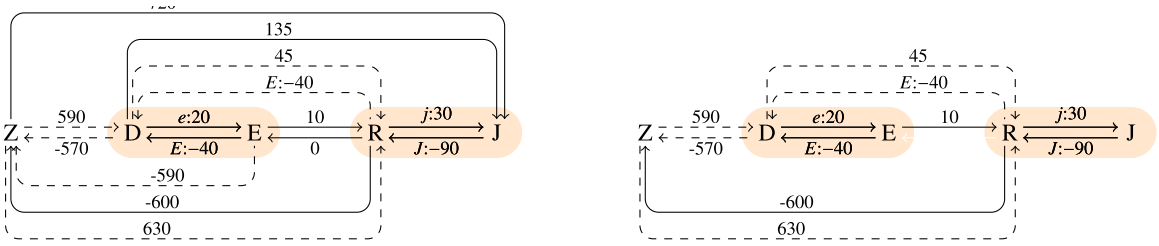
Marking wait edges for removal. At Line 4, $\texttt{minDisp}_{\text{ESTNU}}$ calls the `markWaits` function (Algorithm 27) whose job is to collect the wait edges that are not needed for the ESTNU's dispatchability and hence can be removed from the ESTNU. At Line 3, `markWaits` collects waits dominated by ordinary paths (because $\text{d}(V, A) \leq -v$) or the corresponding UC edge (because $\text{d}(V, C) < 0$). Then, at Lines 5–6, it collects all waits dominated by other waits.

Complexity. The time complexity of $\texttt{minDisp}_{\text{ESTNU}}$ is dominated by the (at most) $k+1$ calls to Johnson's algorithm (in `genStandIns`) whose complexity is $O(mn + n^2 \log n)$, where $m \leq n^2$ is the maximum number of edges in $\mathcal{G}_o$. Therefore, the worst-case complexity of $\texttt{minDisp}_{\text{ESTNU}}$ is $O(kn^3)$.

Correctness. Section A.3 presents a detailed proof of correctness for the `genStandIns` (and $\texttt{minDisp}_{\text{ESTNU}}$) algorithm, leveraging the theoretical insights gained from the analysis of nested diamond structures.

Minimizing the number of edges in the dispatchable Repair-Scenario ESTNU. Fig. 19a recalls the dispatchable ESTNU for the Repair Scenario produced by the $\texttt{FD}_{\text{STNU}}$ algorithm, seen earlier in Fig. 13b. Fig. 19b shows the equivalent dispatchable ESTNU having a *minimal* number of edges that results from applying the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm. The $\texttt{minDisp}_{\text{ESTNU}}$ algorithm removed all of the constraints that are not necessary for incremental execution of the network (e.g., using the $\texttt{RTE}^*$ algorithm). In particular, it removed the original constraints $(\text{Z}, 720, J)$, $(D, 135, J)$, $(R, 0, E)$, and the previously derived constraint $(E, -590, \text{Z})$.

[17] A rigid component is a set of timepoints whose values relative to one another are constrained to be fixed.



(a) Dispatchable ESTNU for Repair Scenario (same as Figure 13b)

(b) Equivalent dispatchable ESTNU minimized by $\texttt{minDisp}_{\text{ESTNU}}$

Fig. 19. Minimizing the number of edges in the dispatchable Repair-Scenario ESTNU.

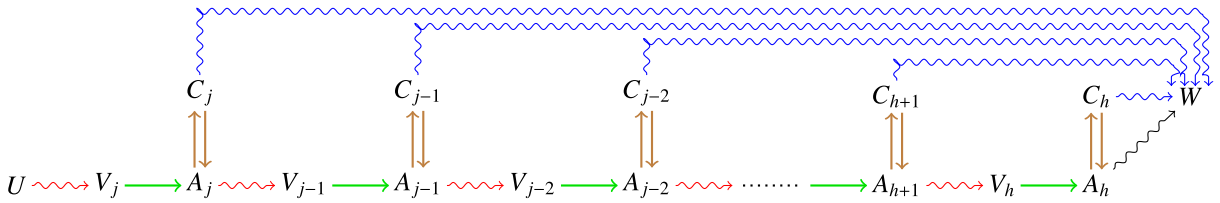


Fig. 20. Canonical form of a nested diamond structure $\mathcal{S}_{uw}$ that determines $d_*(U, W)$, with contingent links in brown, waits in green, negative ordinary edges in red, non-negative ordinary edges in blue, and an ordinary vee-path in black.

7.3. *The* $\texttt{minDisp}^{+}_{\text{ESTNU}}$ *algorithm*

This section introduces $\texttt{minDisp}^{+}_{\text{ESTNU}}$, a new and faster algorithm for solving the MinDispESTNU problem. It modifies $\texttt{minDisp}_{\text{ESTNU}}$ by replacing `genStandIns` with a new algorithm, called `newGenStandIns`, that takes a more focused and efficient approach to exploring nested diamond structures. $\texttt{minDisp}^{+}_{\text{ESTNU}}$ runs in $O(n^3 + k^2 n \log n)$ time. Since $k = O(n)$ is common in applications (e.g., $k \approx n/10$ in some benchmarks [63]), the reduction in worst-case time-complexity is effectively from $O(n^4)$ to $O(n^3 \log n)$.[18]

7.3.1. *Stand-in edges derived from nested diamond structures*

The novel theoretical analysis in Appendix A introduces a rigorous and comprehensive treatment of the *canonical form of nested diamond structures,* providing a foundation for understanding ESTNU-dispatchability that is analogous to how shortest vee-paths provide a foundation for understanding STN-dispatchability. The most important result, Theorem 3, specifies that for each pair of timepoints U and W for which the value of $d_*(U, W) \neq \mathrm{d}(U, W)$ (i.e., for which the value of $d_*(U, W)$ depends on labeled edges), there is a structured combination of ESTNU edges, that can be notated as $\mathcal{S}_{uw}$, having the canonical form illustrated in Fig. 20.[19] This canonical form has the following features:

- There is a sequence of activation timepoints, $A_j, A_{j-1}, A_{j-2}, \ldots, A_{h+1}, A_h$, where each consecutive pair, A_{i+1} and A_i, are connected by a path comprising zero or more ordinary negative edges (shown in red) and one wait edge (shown in green). We refer to such paths as *negOrdWait* paths. (There is also a *negOrdWait* path connecting U with the first activation timepoint A_j.) The ESTNU path from U to W along the bottom of that figure, which includes all of the *negOrdWait* paths, is called the *spine* of the structure.
- Emanating from each contingent timepoint, $C_j, C_{j-1}, \ldots, C_h$, is a path comprising *non-negative* ordinary edges (shown in blue).
- The path from A_h to W (shown in black) is an ordinary shortest vee-path.

[18] Although $\texttt{minDisp}^{+}_{\text{ESTNU}}$ draws from a previously published algorithm, called $\texttt{fastMinDisp}_{\text{ESTNU}}$ [40], a significant oversight in $\texttt{fastMinDisp}_{\text{ESTNU}}$ makes it incomplete (i.e., it can fail to generate all stand-in edges entailed by nested diamonds), making its $O(n^3)$ time complexity irrelevant. That oversight is corrected in $\texttt{minDisp}^{+}_{\text{ESTNU}}$ by introducing a novel replacement for a major algorithmic component, as is discussed below.

[19] The canonical form shown in Fig. 20 assumes that all stand-in edges derived from individual LC, UC and wait edges (cf. the black dashed edges in Fig. 14b) are present in the ESTNU. This assumption is appropriate for analyzing the `genStandIns` and `newGenStandIns` algorithms, both of which begin by inserting those stand-in edges. As a result, the paths of ordinary non-negative edges from each C_i to W (shown in blue in Fig. 20) may include stand-in edges for LC edges; and the paths of ordinary negative edges terminating at each V_i (shown in red) may include stand-in edges for wait or UC edges. Once those stand-in edges are removed from the network, the labeled edges from which they were derived may take their place in the canonical structure.

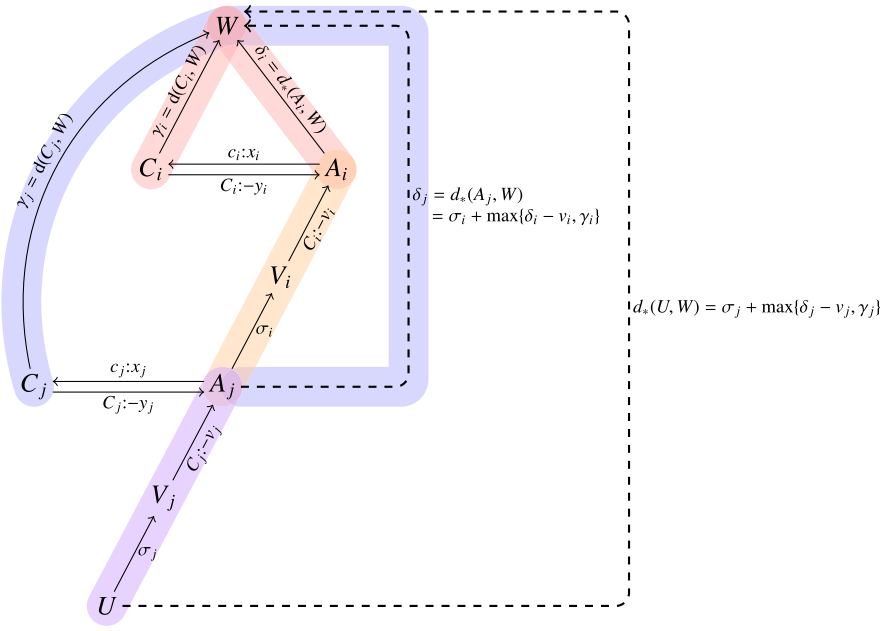


Fig. 21. A canonical structure of nested diamonds with $A_jV_iA_iC_iW$ nested inside $UV_jA_jC_jW$.

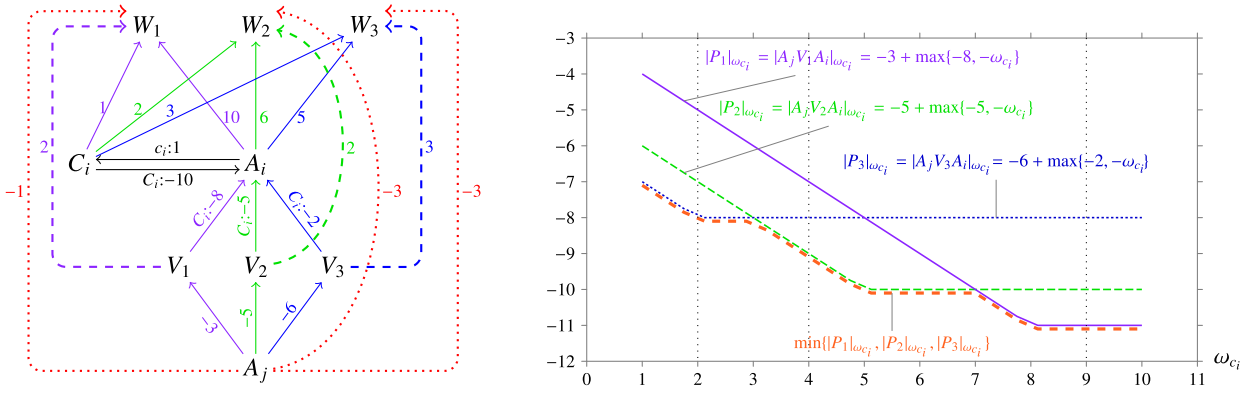


Fig. 22. Three different *negOrdWait* paths from A_j to A_i that are needed to generate stand-in edges from A_j to W_1, W_2 and W_3.

- For each situation ω, there is a path $\mathcal{P}$ from U to W using only edges from $\mathcal{S}_{uw}$ such that $|\mathcal{P}|_\omega \leq d_*(U, W)$; and for the situation where each contingent duration satisfies $C_i - A_i = \delta_i - \gamma_i = d_*(A_i, W) - \mathrm{d}(C_i, W)$, there is a path $\mathcal{P}$ from U to W using only edges from $\mathcal{S}_{uw}$ such that $|\mathcal{P}|_\omega = d_*(U, W)$.

The focus of the rest of this section is on the *spines* of such structures.

Consider any activation timepoints that appear along the spine of *any* canonical structure. Since they are connected by a sequence of *negOrdWait* paths, it follows that *all* of the activation timepoints in the ESTNU that are involved in canonical structures, regardless of which ATPs are involved in which structures, can be put into a strict partial order. That in turn implies that a *single pass* through the activation timepoints is sufficient to generate all of the stand-in edges arising from nested diamonds, thereby avoiding the k passes through the activation timepoints that drives the complexity of `genStandIns`. Toward that end, the rest of this section first presents an algorithm called `getNegOrdWaitInfo` that accumulates information about *negOrdWait* paths connecting pairs of activation timepoints, and then presents the `newGenStandIns` algorithm that exploits this information to more efficiently generate stand-in edges arising from nested diamond structures.

Consider the canonical structure of nested diamonds shown in Fig. 21. In the figure, the inner diamond $A_jV_iA_iC_iW$, with subpaths C_iW, A_iW and $A_jV_iA_i$ shaded red and orange, determines the stand-in edge from A_j to W of length $\delta_j = d_*(A_j, W) = \sigma_i + \max\{\delta_i - v_i, \gamma_i\}$, shown as dashed and shaded purple. The outer diamond $UV_jA_jC_jW$, with sub-

paths C_jW, A_jW (stand-in edge) and UV_jA_j shaded blue and purple, then determines the stand-in edge from U to W of length $d_*(U,W) = \sigma_j + \max\{\delta_j - v_j, \gamma_j\}$. Key features of this nesting include the following. First, the values of $\gamma_j = \mathrm{d}(C_j, W)$, $\sigma_j = \mathrm{d}(U, V_j)$ and $-v_j$ are independent of any nested diamonds. The only value that depends on nested diamonds is $\delta_j = d_*(A_j, W)$ (i.e., the length of the stand-in edge from A_j to W). For this reason, the focus of the `newGenStandIns` algorithm is on generating only the stand-in edges emanating from activation timepoints. And since those activation timepoints are subject to a strict partial order, once that order is known—which is the job of the helper algorithm, `getNegOrdWaitInfo`—those stand-in edges can be efficiently generated by a single-pass exploration of those activation timepoints. Finally, after all of the stand-in edges emanating from activation timepoints have been found, *all* remaining stand-in edges can be computed by a single iteration of the `genStandIns` algorithm.

7.3.2. The `getNegOrdWaitInfo` *algorithm*

Now, as illustrated in Figs. 20–21, for any given timepoints U and W, the canonical structure requires only one *negOrdWait* path between each consecutive pair of activation timepoints, A_j and A_i, along the spine of the structure. However, if A_j and A_i are consecutive activation timepoints in multiple canonical structures that terminate at different timepoints, then, as illustrated on the lefthand side of Fig. 22, those different structures may employ different *negOrdWait* paths from A_j to A_i. The reason is that the different target timepoints (e.g., W_1, W_2 and W_3 in the figure) may generate different values of $\omega_{c_i} = \delta_i - \gamma_i$ due to the different lengths of the paths from C_i to each of the target timepoints, and from A_i to each of the targets.[20] For those different values of ω_{c_i}, different *negOrdWait* paths may provide the shortest routes from A_j to A_i. For example, in the figure, using the target timepoint W_1 generates a value of $\omega_{c_i} = \delta_i - \gamma_i = d(C_i, W_1) - d(A_i, W_1) = 10 - 1 = 9$. For that value of ω_{c_i}, the *negOrdWait* path $A_jV_1A_i$ (shown in purple) is the shortest route from A_j to A_i, and yields the (dotted, red) stand-in edge $(A_j, -1, W_1)$. In contrast, using the target W_2 generates a value of $\omega_{c_i} = \delta_i - \gamma_i = d(C_i, W_2) - d(A_i, W_2) = 6 - 2 = 4$. For that value of ω_{c_i}, the *negOrdWait* path $A_jV_2A_i$ (shown in green) is the shortest route from A_j to A_i, and yields the (dotted, red) stand-in edge $(A_j, -3, W_2)$. Finally, for the target W_3, $\delta_i - \gamma_i = 5 - 3 = 2$, whence $A_jV_3A_i$ (shown in blue) provides the shortest route from A_j to A_i, and yields the (dotted, red) stand-in edge $(A_j, -3, W_3)$.

As suggested by these examples, the shortest route from A_j to A_i that will be needed to generate a stand-in edge for a given target timepoint W generally depends on the value of ω_{c_i}. For different ω_{c_i}, different *negOrdWait* paths from A_j to A_i may have minimal length. For example, the righthand side of Fig. 22 plots the lengths the three different *negOrdWait* paths from A_j to A_i from the ESTNU on the lefthand side of the figure as functions of ω_{c_i}. It also plots the *minimum* of the three functions, shown as dashed and red. At the three values of ω_{c_i} in $\{2, 4, 9\}$, indicated by vertical lines in the plot, the shortest routes from A_j to A_i are given by three different *negOrdWait* paths.

Although, as described above, multiple *negOrdWait* paths may need to be considered between any pair of activation timepoints, the lengths of such paths as functions of the relevant contingent duration have a very restricted form, namely: $\sigma + \max\{-v, -\omega_c\}$, where σ is the length of the subpath of negative ordinary edges, $-v$ is the length of the wait edge, and $\omega_c = C - A$ specifies the relevant contingent duration. For any $\omega_c \geq v$, the length reduces to $L = \sigma - v$. To determine which *negOrdWait* paths must be considered, it is important to determine the *negOrdWait* paths that are not *dominated* by any other *negOrdWait* path.

Definition 13. Let P_1 and P_2 be two *negOrdWait* paths from A_j to A, where A is the activation timepoint for a contingent link (A, x, y, C). P_1 *dominates* P_2 if $|P_1|_{\omega_c} \leq |P_2|_{\omega_c}$ for every $\omega_c \in [x, y]$.

Lemma 1. *Let P_1 and P_2 be two* negOrdWait *paths from A_j to A, where A is the activation timepoint for a contingent link (A, x, y, C). If $|P_1|_{\omega_c} = \sigma_1 + \max\{-v_1, -\omega_c\}$ and $|P_2|_{\omega_c} = \sigma_2 + \max\{-v_2, -\omega_c\}$, then P_1 dominates P_2 if and only if $\sigma_1 - v_1 \leq \sigma_2 - v_2$ and $\sigma_1 \leq \sigma_2$.*

Proof. Suppose that $\sigma_1 - v_1 \leq \sigma_2 - v_2$ and $\sigma_1 \leq \sigma_2$. Then for any $\omega_c \in [x, y]$:

$$\begin{aligned} |P_1|_{\omega_c} &= \sigma_1 + \max\{-v_1, -\omega_c\} = \max\{\sigma_1 - v_1, \sigma_1 - \omega_c\} \\ &\leq \max\{\sigma_2 - v_2, \sigma_2 - \omega_c\} && \text{(Since } \sigma_1 - v_1 \leq \sigma_2 - v_2 \text{ and } \sigma_1 \leq \sigma_2\text{)} \\ &= \sigma_2 + \max\{-v_2, -\omega_c\} = |P_2|_{\omega_c}. \end{aligned}$$

On the other hand, if $\sigma_1 > \sigma_2$, then for $\omega_c = x$, $|P_1|_x = \sigma_1 - x > \sigma_2 - x = |P_2|_x$. Or if $\sigma_1 - v_1 > \sigma_2 - v_2$, then for $\omega_c = y$, $|P_1|_y = \sigma_1 - v_1 > \sigma_2 - v_2 = |P_2|_y$. □

If for each *negOrdWait* path determined by σ and $-v$, we let L notate the value $\sigma - v$, then Lemma 1 can be restated as follows: P_1 dominates P_2 if and only if $L_1 \leq L_2$ and $\sigma_1 \leq \sigma_2$. Next, let $P_1, \ldots, P_g$ be a list of all of the *negOrdWait* paths from A_j to A, sorted into non-increasing order, first by their L values and second by their σ values; and let $P(\omega_c) =$

[20] The `fastMinDisp`$_{\text{ESTNU}}$ algorithm [40] implicitly (and incorrectly) assumed that for a pair of timepoints A_j and A_i, a *single negOrdWait* path between them would suffice across *all* nested diamond structures in which they both participated. That oversight makes the algorithm incomplete.

$\min\{|P_1|_{\omega_c}, \ldots, |P_g|_{\omega_c}\}$ be the minimum of those path lengths as a function of ω_c, shown as red and dashed in Fig. 22. Then a single pass through the sorted list can determine the subintervals of $[x, y]$ over which the dominating *negOrdWait* paths equal the minimum $P(\omega_c)$.

Consider the first two *negOrdWait* paths in the sorted list, P_1 and P_2, for which $L_1 \leq L_2$ must hold. Now, if $\sigma_1 \leq \sigma_2$, P_1 dominates P_2 and, so, P_2 can be discarded. Otherwise, $\sigma_1 > \sigma_2$ which implies that $|P_2|_{\omega_c} < |P_1|_{\omega_c}$ over some subinterval of the form $[x, x_1)$ and $|P_1|_{\omega_c} < |P_2|_{\omega_c}$ over $(x_1, y]$, with equality at $\omega_c = x_1$. The crossover value, x_1, is determined by the intersection of the horizontal branch of the plot of $|P_2|_{\omega_c}$ and the diagonal branch of $|P_1|_{\omega_c}$, in other words, where $L_2 = \sigma_1 - \omega_c$ (i.e., where $\omega_c = \sigma_1 - L_2$). For example, in the plot in Fig. 22, $\omega_{c_i} = -3 - (-10) = 7$, which is where the purple line (i.e., $|P_1|_{\omega_{c_i}}$) intersects the green dashed line (i.e., $|P_2|_{\omega_{c_i}}$). Subsequent *negOrdWait* paths in the sorted list can be processed in the same fashion, yielding a sequence of adjacent subintervals that together cover the interval $[x, y]$. In the figure, the subintervals are $[1, 3]$, $[3, 7]$ and $[7, 10]$, where P_3, P_2 and P_1 dominate, respectively.

Algorithm 28: `getNegOrdWaitInfo`: find shortest *negOrdWait* paths between pairs of activation timepoints.

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E}_o, \mathcal{E}_l, \mathcal{E}_u, \mathcal{E}_w)$, an ESTNU graph; and f, a potential function for $(\mathcal{T}, \mathcal{E}_o)$
Output: *negOrdWaits*, a k-by-k array where each element (A_j, A_i) is a set of quadruples of the form (L, σ, lb, ub) representing the *negOrdWait* paths from A_j to A_i and the subintervals over which they are dominant.

```
1  G⁻ := (T, E_o⁻), where E_o⁻ = {(U, δ, V) ∈ E_o | δ < 0}
2  d⁻ := k-by-n array returned by Johnson's alg on G⁻, using f as a potential function, but only using ATPs as source TPs
3  M := a k-by-k array, initially all entries ∅
   /* Initialize each M[A_j][A_i] entry to include all negOrdWait paths from A_j to A_i */
4  foreach (V, C:−v, A) ∈ E_w do
5      foreach A_j ∈ ActTPs(G) do
6          σ := d⁻(A_j, V)                                   // σ is either negative or ∞
7          if σ < 0 then M[A_j][A].push((σ − v, σ))          // σ − v = L = length of negOrdWait path
   /* Each negOrdWait[A_j][A_i] will accumulate info about which negOrdWait paths are dominant over which subintervals */
8  negOrdWait := a k-by-k array, initially all entries ∅
9  foreach i ∈ {1, 2, ..., k} do
10     foreach j ∈ {1, 2, ..., k} do
11         if M[A_j][A_i] ≠ ∅ then
               /* Determine the subintervals of [x_i, y_i] over which the negOrdWait paths in M[A_j][A_i] are dominant */
12             M[A_j][A_i] := sort M[A_j][A_i] into non-decreasing order, first w.r.t. L values, second w.r.t. σ values
13             (L, σ) := M[A_j][A_i].pop()                   // (L, σ) nec. dominant over some subinterval since it has min L and min σ values
14             ub := y_i                                     // y_i is the upper bound for the contingent duration associated with A_i
15             foreach (L′, σ′) ∈ M[A_j][A_i] do
16                 if σ′ < σ then                            // (L′, σ′) necessarily dominant over some subinterval
17                     lb := σ − L′                          // lb is the value of ω_{c_i} where negOrdWait paths determined by (L, σ) and (L′, σ′) intersect
18                     negOrdWaits[A_j][A_i].push((L, σ, lb, ub))   // Path determined by (L, σ) is minimal over [lb, ub]
19                     ub := lb                              // Prepare for next iteration
20                     (L, σ) := (L′, σ′)
21             negOrdWaits[A_j][A_i].push((L, σ, x_i, ub))   // (L, σ) dominant over leftmost subinterval [x_i, ub]
22 return negOrdWaits
```

Pseudocode for the `getNegOrdWaitInfo` algorithm is given in Algorithm 28. Lines 1 and 2 use a restricted version of Johnson's algorithm to compute the distances of all ordinary paths whose source node is an activation timepoint, and whose edges all must be negative. It uses the potential function f, which is provided as an input to `getNegOrdWaitInfo`, to guide the exploration of such paths. Next, Lines 3–7 create a k-by-k matrix $\mathcal{M}$ and then populate each entry $\mathcal{M}[A_j][A_i]$ with pairs of the form (L, σ) that represent *all* of the *negOrdWait* paths from A_j to A_i. The *negOrdWait* paths are accumulated by iterating through all of the ESTNU's wait edges and activation timepoints. In particular, for each wait edge $(V, C{:}-v, A)$ and each activation timepoint A_j, the length of the shortest *negOrd* path from A_j to V is given by $\sigma = d(A_j, V)$, and the corresponding *negOrdWait* path is specified by (σ, L), where $L = \sigma - v$.

Next, the doubly nested `for` loops at Lines 9–21 compute, for each pair A_j and A_i of activation timepoints, the sets of *undominated negOrdWait* paths from A_j to A_i and the subintervals of $[x_i, y_i]$ over which they are dominant. First, at Line 12, the list of all *negOrdWait* paths from A_j to A_i is sorted into non-decreasing order, first by L values, and second by σ values. Next, at Line 13, the first (L, σ) pair is popped from the sorted list. The *negOrdWait* path represented by that pair is necessarily dominant (i.e., its length is the minimum among all *negOrdWait* paths from A_j to A_i) over some subinterval of the form (lb, y_i), where y_i is the maximum duration of the i^{th} contingent link. The nested `for` loop at Lines 15–20 walks through the remaining *negOrdWait* paths to discard any that are dominated over the entire interval $[x_i, y_i]$ and, for the rest, compute the subintervals over which they are undominated (i.e., have minimum length), as described earlier. For each pair (L, σ) that represents a *negOrdWait* path that is undominated over some subinterval $[a, b]$, an entry (L, σ, a, b) is

accumulated in *negOrdWaits*$[A_j][A_i]$ (cf. Line 20). After all such entries are accumulated, the *negOrdWaits* array is returned (cf. Line 22).

Algorithm 29: `newGenStandIns`: Compute the stand-in edges arising from nested diamonds.

Input: $\mathcal{G} = (\mathcal{T}_x \cup \mathcal{T}_c, \mathcal{E}_o \cup \mathcal{E}_l \cup \mathcal{E}_u \cup \mathcal{E}_w)$, a dispatchable ESTNU
Output: $(\mathcal{E}_o^{si}, d)$, where $\mathcal{E}_o^{si}$ is the set of *all* stand-in edges, and d is the updated distance matrix.

```
1  L := the contingent links associated with G
2  E_o^si := getInitStandins(G)          // Collect stand-in edges for individual labeled edges; and fix weak or misleading waits
3  (d, f) := johnson(T, E_o ∪ E_o^si)    // Compute distance matrix d for the ordinary ESTNU edges; return potential function f
4  negOrdWaits := getNegOrdWaitInfo(G, f)
5  readyToGo := ∅                        // A list of activation timepoints ready for processing
6  numUnprocessed := (0, ..., 0)         // For each activation timepoint, the number of its unprocessed children
7  foreach j ∈ {1, 2, ..., k} do
8      foreach i ∈ {1, 2, ..., k} do
9          if negOrdWaits[A_j][A_i] ≠ ∅ then numUnprocessed[j] := numUnprocessed[j] + 1
10     if numUnprocessed[j] == 0 then readyToGo.push(A_j)
11 while readyToGo ≠ ∅ do
12     A_j := readyToGo.pop()            // Contingent link for A_j is (A_j, x_j, y_j, C_j)
13     foreach i ∈ {1, 2, ..., k} do
14         if negOrdWaits[A_j][A_i] ≠ ∅ then
15             foreach W ∈ T \ {A_i, C_i, A_j, C_j} do
16                 γ_i = d(C_i, W);  δ_i = d(A_i, W);  ω_i := δ_i − γ_i
17                 if ω_i ∈ (x_i, y_i] then   // ω_i specifies projection where maximum SVP occurs
                       /* fetchLS does a binary search on entries in negOrdWaits[A_j][A_i], looking for the (L, σ, a, b) entry where ω_i ∈ [a, b]
                          (i.e., the path determined by (L, σ) is minimal at ω_i). */
18                     (L_i, σ_i) := fetchLS (negOrdWaits[A_j][A_i], ω_i)
19                     v_i = σ_i − L_i;  newVal := σ_i + max{γ_i, δ_i − v_i}
20                     if newVal < d(A_j, W) then
21                         d(A_j, W) := newVal          // Changes to d will be propagated later on
22                         E_o^si := E_o^si ∪ {(A_j, newVal, W)}   // Collect stand-in edge
23     foreach g ∈ {1, 2, ..., k} do
24         if negOrdWaits[A_g][A_j] ≠ ∅ then
25             numUnprocessed[A_g] := numUnprocessed[A_g] − 1
26             if numUnprocessed[A_g] == 0 then readyToGo.push(A_g)
27 (E_o^si, _) := getDiamondStandins(T, L, E_w, E_o^si, d)   // Collect stand-in edges entailed by outermost diamonds
28 d := johnson(T, E_o ∪ E_o^si)          // Update distance matrix d to accommodate all stand-in edges
29 return (E_o^si, d)
```

7.3.3. *The* `newGenStandIns` *algorithm*

This section presents `newGenStandIns` (Algorithm 29). It uses the entries in the *negOrdWaits* array computed by `getNegOrdWaitInfo` to more efficiently generate all of the stand-in edges arising from nested diamond structures. Its time-complexity is $O(n^3 + k^2 n \log n)$, a significant improvement over the $O(kn^3)$-time complexity of `genStandIns`.

Pseudocode for `newGenStandIns` is given in Algorithm 29. Like `genStandIns`, it first calls `getInitStandins` at Line 2 to fix all weak or misleading wait edges, and to generate stand-in edges associated with individual LC, UC and wait edges; and then, at Line 3, Johnson's algorithm to compute the distance matrix for the ordinary edges in the ESTNU. However, whereas `genStandIns` calls Johnson's algorithm on each of up to k iterations of its main loop, `newGenStandIns` calls Johnson only twice: once at Line 3, and once at Line 28, after all of the stand-in edges have been generated. The initial call to Johnson is also modified to return the potential function f that it uses to re-weight the ordinary edges, thereby enabling that same potential function to be passed as an input to `getNegOrdWaitInfo`, at Line 4. The information contained in the k-by-k *negOrdWaits* array returned by `getNegOrdWaitInfo` represents all of the *negOrdWait* paths connecting each pair of activation timepoints that could be adjacent in the spine of one or more canonical structures of nested diamonds.

Next, Lines 5–10 initialize two data structures that ensure efficient processing by the rest of the algorithm: *readyToGo* and *numUnprocessed*. *readyToGo* is a list of the activation timepoints that are ready to process. Since the activation timepoints form a strict partial order, this list is initially populated by those A_j from which no *negOrdWait* paths emanate (i.e., those A_j such that for each A_i, *negOrdWaits*$[A_j][A_i] = \emptyset$). (For convenience, if there is a *negOrdWait* path from A_j to A_i, we call A_j a *parent* of A_i, and A_i a child of A_j. Therefore, *readyToGo* initially contains all A_j that have no children.) The vector,

numUnprocessed, keeps track of how many *unprocessed children* each activation timepoint has. Each *numUnprocessed*[A_j] entry is initialized at Lines 5–9. Later on, as each activation timepoint is processed, its parent's entry in *numUnprocessed* is decremented (cf. Line 25). If *numUnprocessed*[A_g] ever reaches zero, then all of A_g's children must have been processed, in which case, A_g is then pushed onto the *readyToGo* list (cf. Line 26). In this way, each timepoint A_g is processed only after all of its children have been processed, thereby respecting the strict partial order among the activation timepoints.

The main processing of `newGenStandIns` is carried out by the `while` loop at Lines 11–26. Each iteration processes one activation timepoint A_j, popped off *readyToGo* at Line 12. Then, for each child A_i of A_j, and for each timepoint W, the nested `for` loops at Lines 13–22 check whether a stand-in edge from A_j to W can be generated using a *negOrdWait* path from A_j to A_i. As dictated by Theorem 3 in Appendix A, a stand-in edge can be generated if $\omega_i = \delta_i - \gamma_i = d_*(A_i, W) - \mathrm{d}(C_i, W) \in (x_i, y_i]$, where (A_i, x_i, y_i, C_i) is the contingent link whose activation timepoint is A_i. In this context, the value of $d_*(A_i, W)$ is represented at Line 16 as $\mathrm{d}(A_i, W)$ because $d_*(A_i, W)$ equals either the original value of $\mathrm{d}(A_i, W)$ or the length of some stand-in edge from A_i to W generated by a prior iteration of the `while` loop. If $\omega_i \in (x_i, y_i]$, the ordered sequence of *negOrdWait* paths from A_j to A_i that are stored in *negOrdWaits*[A_j][A_i] must be searched to find the one that is undominated for $\omega_i = \delta_i - \gamma_i$. That is handled by calling `fetchLS` at Line 18, which returns a pair (L_i, σ_i) representing the relevant *negOrdWait* path. (Since `fetchLS` does a simple binary search over an ordered array of *negOrdWait* quadruples, we do not provide pseudocode for it.) Next, at Line 19, the length of a potential stand-in edge from A_j to A_i is computed according to Theorem 3: *newVal* $:= \sigma_i + \max\{\delta_i - v_i, \gamma_i\}$, where $v_i = \sigma_i - L$. If that length is shorter than or equal to the length of any pre-existing ordinary edge from A_j to A_i, then a new stand-in edge is accumulated and the value of $\mathrm{d}(A_j, W)$ is updated (cf. Lines 20–22).

After all of A_j's children have been processed, then, as mentioned earlier, the *numUnprocessed* value for each of A_j's parents is updated and, if any of A_j's parents thereby become ready for processing, they are pushed onto the *readyToGo* list (cf. Lines 23–26).

Once all activation timepoints have been fully processed (i.e., after the `while` loop is finished, at Line 27), all stand-in edges emanating from activation timepoints to any other timepoints in the network have been computed. Then, for *any* timepoints U and W, the *portion* of the spine of a canonical structure corresponding to $d_*(U, W)$ has been effectively computed from the first activation timepoint, A_j, all the way to W. (Recall Fig. 21.) The only remaining portion is the *negOrdWait* path from U to A_j. The stand-in edges corresponding to such *outermost* nestings of diamond structures are then generated at Line 27 by calling `getDiamondStandins`, which does a *single* iteration of the original `genStandIns` algorithm. After that, *all* stand-in edges have been generated, whereupon a final call to Johnson's algorithm at Line 28 updates the distance matrix for the ordinary ESTNU edges, which now include all stand-in edges.

7.3.4. Complexity of `newGenStandIns`

The $\texttt{minDisp}^{+}_{\text{ESTNU}}$ algorithm is the same as $\texttt{minDisp}_{\text{ESTNU}}$, except that the `genStandIns` helper has been replaced by the `newGenStandIns` algorithm presented above. The complexity of `newGenStandIns` derives from the complexity of (1) `getNegOrdWaitInfo`; (2) its main `while` loop; and (3) its two calls to Johnson's algorithm.

The complexity of `getNegOrdWaitInfo` is driven by its doubly nested `for` loops, each of which involves k iterations and an $O(n \log n)$-time sorting of at most n *negOrdWait* pairs, for an overall complexity of $O(k^2 n \log n)$.

The `while` loop of `newGenStandIns` does k iterations, one for each activation timepoint in the network. Each iteration uses doubly nested `for` loops, with k and n iterations respectively. Inside those nested `for` loops, the call to `fetchLS`, which uses a binary search over at most n *negOrdWait* quadruples, has a complexity of $O(\log n)$. Therefore, the overall complexity of the `while` loop is $O(k^2 n \log n)$.

Since the complexity of Johnson's algorithm is $O(mn + n^2 \log n)$, where $m = O(n^2)$ is the maximum number of ordinary edges, including all stand-in edges, the two calls to Johnson's algorithm contribute $O(n^3 + n^2 \log n) = O(n^3)$.

Therefore, the overall complexity of `newGenStandIns` is $O(n^3 + k^2 n \log n)$.

The complexity of steps 2, 3 and 4 of $\texttt{minDisp}^{+}_{\text{ESTNU}}$, which we do not change, is dominated by the call to the STN-dispatchability algorithm on at most n^2 edges, which is $O(n^3)$. So the overall complexity of the new $\texttt{minDisp}^{+}_{\text{ESTNU}}$ algorithm is $O(n^3 + k^2 n \log n)$.

We conjecture that, in practice, the numbers of *negOrdWait* quadruples in each *negOrdWaits*[A_j][A_i] entry will be *very* small. If so, that would make the effective complexity of $\texttt{minDisp}^{+}_{\text{ESTNU}}$ $O(n^3)$.

8. Empirical evaluations

This section summarizes the experimental evaluations that compare the performance and output quality of the algorithms discussed in this article. All algorithms were implemented in Java and are publicly available as part of the CSTNU Tool framework [64]. The tool enables users to create different kinds of temporal constraint networks and automatically verify properties such as dynamic controllability, dispatchability, or consistency, depending on the network type. For STNUs, it provides implementations of algorithms for dynamic controllability, dispatchability, detection of semi-reducible negative cycles, and real-time execution. As an illustration, the screenshot in Fig. 23 shows the CSTNU Tool after running the `findSRNC` algorithm on the STNU from Fig. 7. The left-hand side displays the editable initial network, while the right-hand side highlights (in red) the semi-reducible negative cycle found by `findSRNC`. The status bar above the network on the right

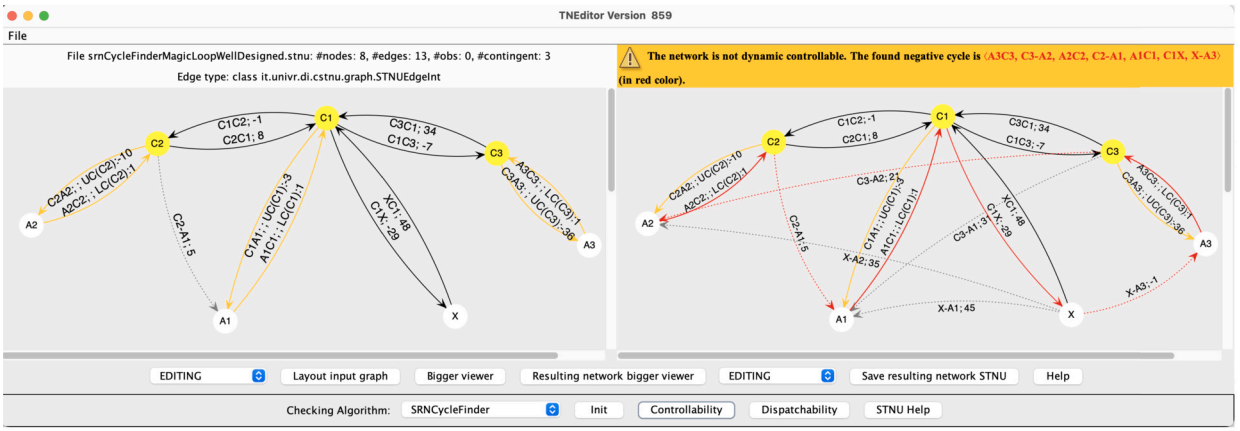


Fig. 23. A screenshot of the `findSRNC` algorithm in action.

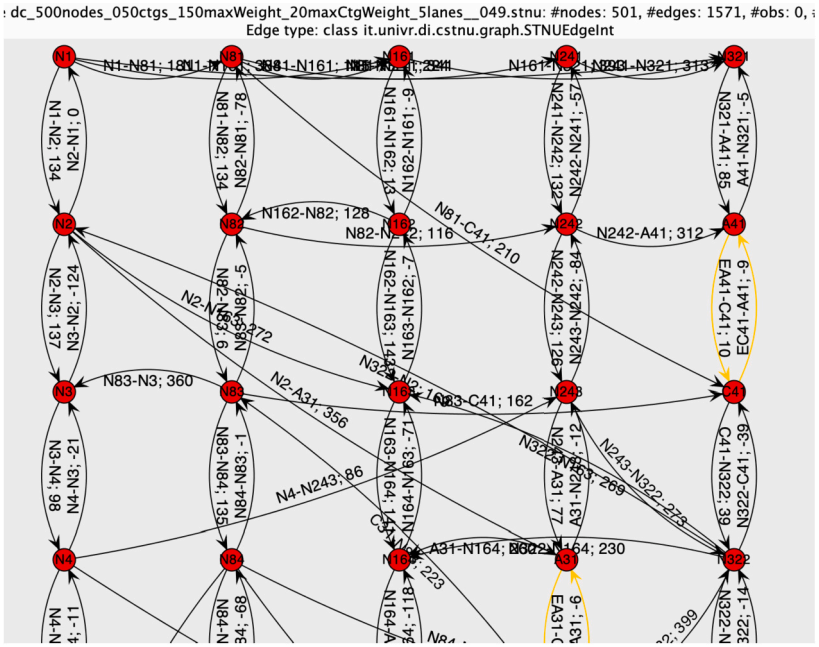


Fig. 24. A partial view of a typical randomly generated STNU, with contingent links in yellow.

gives a brief summary of the `findSRNC` result. The extended output, including the fully expanded semi-reducible negative cycle, is saved to the execution log.

Each algorithm was evaluated using the STNU benchmark published by Posenato [63]. The benchmark was generated using the CSTNU Tool's STNU random generator, which can create instances with various topologies adjustable via input parameters. The public benchmark comprises 1000 instances, all using the *worker-lanes* topology, which simulates worker lanes in business process modeling [61]. In this topology, the set of contingent links is divided into five lanes, with each lane representing a sequence of tasks that must be executed by an agent. The contingent links within each lane are interspersed with ordinary constraints that specify delays between the end of one task and the start of the next. Additionally, constraints between nodes in different lanes represent temporal coordination among tasks executed by different agents. Typically, such constraints involve nodes on different lanes that are at a similar distance from the start of their respective lanes. Fig. 24 shows a portion of a random STNU with 500 nodes and 50 contingent links in a 5-worker-lane topology.

For each possible number of nodes $n \in \{500, 1000, 1500, 2000, 2500\}$, the benchmark contains 200 DC instances and 200 non-DC instances, each having $k = n/10$ contingent links. Table 6 summarizes the parameters used to generate the instances of the STNU benchmarks. With these parameters, each node has two incoming and two outgoing edges within its lane, and an average of 2.56 incident edges (excluding activation timepoints) representing temporal coordination with nodes in other

Table 6
Parameters used to generate instances for the STNU benchmarks.

Number of nodes, n	$n \in \{500, 1000, 1500, 2000, 2500\}$
Number of lanes	5
Number of contingent links, k	$k = n/10$ (hence, $k = O(n)$)
Maximum absolute weight of ordinary edges	150
Maximum range of contingent durations	$[1, 20]$
Probability of constraints among nodes in different lanes	0.40

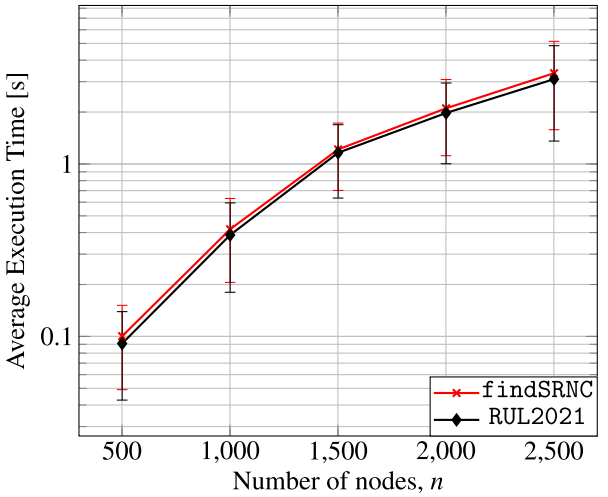


Fig. 25. Average execution times of `findSRNC` and `RUL2021` vs. network size.

Table 7
Experimental results for `findSRNC` algorithm.

n	Avg num edges in SRN cycles	% Simple SRN cycles	Avg num edges in expanded SRN cycles
500	8.0	63%	14.1
1000	7.9	64%	14.4
1500	7.6	74%	15.3
2000	8.5	72%	14.0
2500	8.6	63%	14.4

lanes. Temporal coordination constraints are chosen to avoid introducing negative cycles between pairs of nodes. Therefore, the average number of edges m is $6.56n - 2.56k - 10$, and hence $m = O(n)$.

All of the experiments were executed on OpenJDK JVM 21 configured with a 16 GB heap (parameters `-Xmx16G` and `-Xms16G`) on a Linux machine equipped with two AMD Opteron 4334 processors (3.1 GHz) and 64 GB of RAM.

8.1. Empirical evaluation of the `findSRNC` algorithm

This section presents an empirical evaluation of the `findSRNC` algorithm's execution time and a quality assessment of the non-DC instances from the main benchmark. The objectives of the evaluation were twofold: (1) to confirm that the execution times of `findSRNC` and `RUL2021` are comparable; and (2) to highlight the characteristics of the SRN cycles in non-DC instances.

We considered the non-DC instances from the main benchmark for each $n \in \{500, 1000, 1500, 2000, 2500\}$. For each sub-benchmark (i.e., for each n), we used the first 100 instances. For each instance, both algorithms confirmed the non-DC status, while `findSRNC` also returned an SRN cycle.

Fig. 25 shows the average execution times of the two algorithms for each sub-benchmark. These results indicate that computing the SRN cycle does not introduce significant computational overhead. More importantly, by analyzing the cycles computed by `findSRNC`, we can, for the first time, evaluate the characteristics of the non-DC instances in the benchmark. Table 7 presents the statistics we accumulated for each sub-benchmark. The data shows that, for each n, the average number of edges in the SRN cycle is relatively small (less than 9), and most instances exhibit a *simple* SRN cycle (i.e., an SRN cycle with no annotated bypass edges) comprising only edges present in the input STNU.

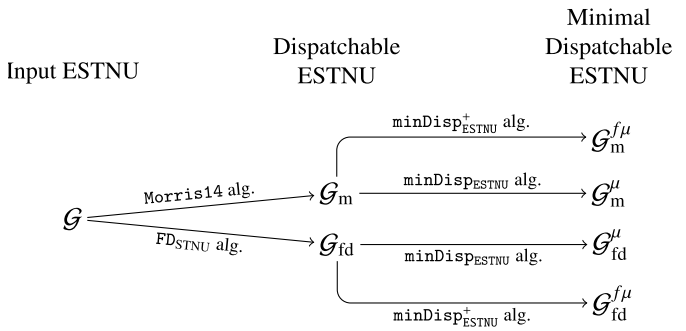


Fig. 26. Evaluating the $\mathtt{minDisp}_{\mathrm{ESTNU}}$ and $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ algorithms.

When `findSRNC` outputs a *non-simple* SRN cycle (i.e., an SRN cycle that includes annotated bypass edges), it does so in a compact form. However, we also computed the *fully expanded* version of each cycle by recursively replacing each bypass edge with the annotated path from which it was derived. The average number of edges in the expanded SRN cycles reached a maximum of 16 across the benchmarks, revealing that an SRN cycle can involve more edges from the original STNU than one might suspect from the compact version. Finally, we verified that *no* instance leads to an expanded SRN cycle with *any* repeated edges.

8.2. Empirical evaluation of the $\mathtt{minDisp}_{\mathrm{ESTNU}}$ and $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ algorithms

This section evaluates the performance of the $\mathtt{minDisp}_{\mathrm{ESTNU}}$ and $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ algorithms for solving the MinDispESTNU problem (i.e., for computing μESTNUs). The objectives were twofold: (1) to assess the increase in computational cost required to generate μESTNUs; and (2) to demonstrate the magnitude of the reduction in the numbers of edges in minimal networks. The results presented in this section are new and different from the results presented earlier [39] because in this work we optimized the implementation of the $\mathtt{minDisp}_{\mathrm{ESTNU}}$ algorithm to prevent some timeouts encountered in previous work. The optimization was achieved by adopting a more efficient representation of the distance matrix, which is a key data structure in the algorithm. The new implementation introduces a specialized data structure for the distance matrix that retains the flexibility to access distances between nodes using their names, as required by the CSTNU Tool framework, while ensuring $O(1)$ time complexity for adding, retrieving, or modifying each distance.

The evaluation was conducted using DC instances from the main benchmark, 30 for each value of n in $\{500, 1000, 1500, 2000\}$. As illustrated in Fig. 26, each DC STNU $\mathcal{G}$ was first pre-processed by the `Morris14` and $\mathtt{FD}_{\mathrm{STNU}}$ dispatchability algorithms to generate equivalent dispatchable ESTNUs, $\mathcal{G}_{\mathrm{m}}$ and $\mathcal{G}_{\mathrm{fd}}$. Next, $\mathcal{G}_{\mathrm{m}}$ and $\mathcal{G}_{\mathrm{fd}}$ were fed as input to the $\mathtt{minDisp}_{\mathrm{ESTNU}}$ algorithm to check that the resulting μESTNUs, $\mathcal{G}^{\mu}_{\mathrm{m}}$ and $\mathcal{G}^{\mu}_{\mathrm{fd}}$, were identical. Finally, the dispatchable ESTNUs, $\mathcal{G}_{\mathrm{m}}$ and $\mathcal{G}_{\mathrm{fd}}$, were fed as input to $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ (1) to confirm that the resulting μESTNUs, $\mathcal{G}^{f\mu}_{\mathrm{m}}$ and $\mathcal{G}^{f\mu}_{\mathrm{fd}}$, were identical to $\mathcal{G}^{\mu}_{\mathrm{m}}$ and $\mathcal{G}^{\mu}_{\mathrm{fd}}$ (i.e., the ones generated by $\mathtt{minDisp}_{\mathrm{ESTNU}}$); and (2) to demonstrate the average execution times for $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$.

In most cases, the minimized ESTNUs were identical. However, for a few instances, the minimized ESTNUs differed slightly in their edges. This discrepancy was attributed to simultaneity constraints among pairs of timepoints in rigid components, which can lead to trivially different, but equivalent minimized ESTNUs.

Surprisingly, during the execution of $\mathtt{minDisp}_{\mathrm{ESTNU}}$, we observed that *no instances* from the considered benchmarks contain any nested diamond structures. Consequently, there were no opportunities for the $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ algorithm to outperform $\mathtt{minDisp}_{\mathrm{ESTNU}}$. Nonetheless, the execution times of $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ are reported to determine the impact of any overhead associated with the `getNegOrdWaitInfo` helper.

Fig. 27 shows the average numbers of edges in the original STNUs (black), the dispatchable ESTNUs generated by `Morris14` (red) and $\mathtt{FD}_{\mathrm{STNU}}$ (teal), and the minimal dispatchable ESTNUs produced by $\mathtt{minDisp}_{\mathrm{ESTNU}}$ (dotted green) and $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ (dashed blue). (The dotted green and dashed blue lines in the figure are completely overlapping and, hence, difficult to distinguish.) The error bars denote 95% confidence intervals, which are scarcely visible due to the minimal standard deviations. The findings reveal that the average numbers of edges in the minimized networks are approximately two orders of magnitude smaller than the average numbers of edges in the ESTNUs generated by `Morris14`, and about one order of magnitude smaller than in the ESTNUs generated by $\mathtt{FD}_{\mathrm{STNU}}$. Since the number of edges in dispatchable networks directly impacts the performance of real-time execution algorithms, these results demonstrate that $\mathtt{minDisp}_{\mathrm{ESTNU}}$ and $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ generate dispatchable networks that can be more efficiently executed. We also confirmed that $\mathtt{minDisp}_{\mathrm{ESTNU}}$ and $\mathtt{minDisp}^{+}_{\mathrm{ESTNU}}$ determine the same minimal networks.

Fig. 28 illustrates the computational cost associated with generating minimal dispatchable ESTNUs. The lower two (red and teal) lines show the average execution times for `Morris14` and $\mathtt{FD}_{\mathrm{STNU}}$ to generate equivalent, but non-minimized dispatchable networks, $\mathcal{G}_{\mathrm{m}}$ and $\mathcal{G}_{\mathrm{fd}}$. The upper four (green, blue, violet and orange) lines show the average execution times for generating equivalent dispatchable networks having minimal numbers of edges, obtained by applying $\mathtt{minDisp}_{\mathrm{ESTNU}}$

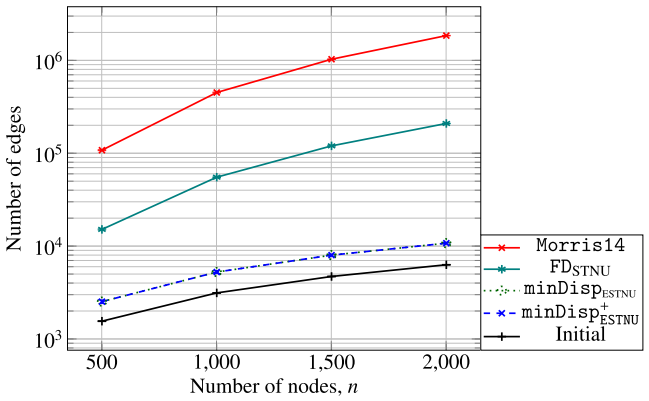


Fig. 27. Number of edges in the dispatchable networks generated by the `Morris14`, FD_{STNU} and $\text{minDisp}_{\text{ESTNU}}$ algorithms.

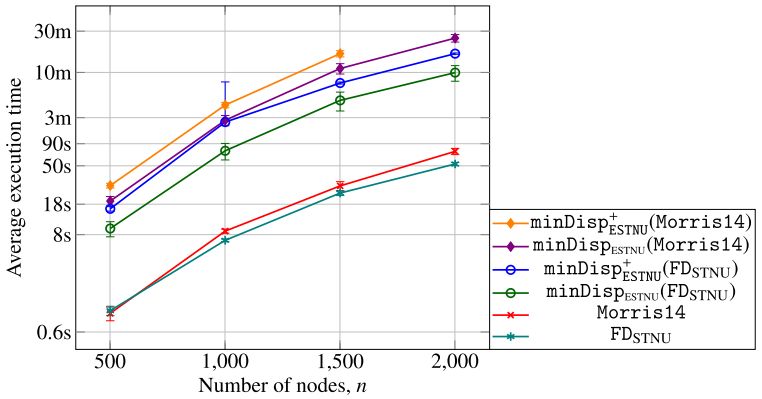


Fig. 28. $\text{minDisp}_{\text{ESTNU}}$ and $\text{minDisp}^{+}_{\text{ESTNU}}$ performance versus network size when no instance contains nested diamond structures.

or $\text{minDisp}^{+}_{\text{ESTNU}}$ to $\mathcal{G}_{\text{m}}$ or $\mathcal{G}_{\text{fd}}$. The results also highlight the significant impact of the number of edges on computation time. For example, when the dispatchable ESTNUs are generated by `Morris14`, which results in greater numbers of edges, and then fed into the $\text{minDisp}_{\text{ESTNU}}$ algorithm, the average execution time, labeled $\text{minDisp}_{\text{ESTNU}}$ (`Morris14`) (violet line) in the figure, is significantly greater than the average execution time when the input instances are generated by FD_{STNU}, labeled $\text{minDisp}_{\text{ESTNU}}$ (FD_{STNU}) (green line) in the figure. In prior work [39], the difference between the execution time of $\text{minDisp}_{\text{ESTNU}}$ (`Morris14`) and $\text{minDisp}_{\text{ESTNU}}$ (FD_{STNU}) was an order of magnitude greater and the execution time of $\text{minDisp}_{\text{ESTNU}}$ (`Morris14`) exceeded the 30-minute timeout for $n = 2000$. However, here, the difference is smaller (e.g., for $n = 1500$, the average execution time of $\text{minDisp}_{\text{ESTNU}}$ (`Morris14`) is 649s, compared to only 256s for $\text{minDisp}_{\text{ESTNU}}$ (FD_{STNU})). We attribute this difference to the improvement in the implementation of the distance matrices used in $\text{minDisp}_{\text{ESTNU}}$ algorithm. This improvement also avoided timeouts for $\text{minDisp}_{\text{ESTNU}}$(`Morris14`) on instances where $n = 2000$ that occurred in the previous version of the algorithm.

Despite its name, the execution times for our $\text{minDisp}^{+}_{\text{ESTNU}}$ algorithm were greater than those for $\text{minDisp}_{\text{ESTNU}}$ across all instances. We attribute this to the overhead of computing alternative *negOrdWait* paths by the `getNegOrdWaitInfo` algorithm, that worsens the performance of the algorithm for instances where there are no nested diamond structures—which is the case across this benchmark. The results can be summarized as follows.

1. The execution time of $\text{minDisp}^{+}_{\text{ESTNU}}$ (`Morris14`) (orange line) is $[1.48, \ldots, 1.59]$ times greater than the execution time of $\text{minDisp}_{\text{ESTNU}}$ (`Morris14`) (violet line). For instances having 2000 nodes, the execution time of $\text{minDisp}^{+}_{\text{ESTNU}}$ (`Morris14`) exceeded the 30-minute timeout for many instances and, therefore, its average execution time is not shown in the plot.
2. The execution time of $\text{minDisp}^{+}_{\text{ESTNU}}$ (FD_{STNU}) (blue line) is $[2.18, \ldots, 2.30]$ times greater than the execution time of $\text{minDisp}_{\text{ESTNU}}$ (FD_{STNU}) (green line).
3. The execution time of $\text{minDisp}^{+}_{\text{ESTNU}}$ (FD_{STNU}) (blue line) is smaller than the execution time of $\text{minDisp}_{\text{ESTNU}}$ (`Morris14`) (violet line) for any instance.

Table 8
Parameters used to generate instances with one quadruply-nested diamond structure.

Number of nodes, n	$n \in \{500, 1000, 1500, 2000, 2500\}$
Number of lanes	5
Number of contingent links, k	$k = n/10$ (hence, $k = O(n)$)
Maximum absolute weight of ordinary edges	150
Maximum range of contingent durations	$[1, 20]$
Probability of constraints among nodes in different lanes	0.40
Number of quadruply-nested diamond structures	1

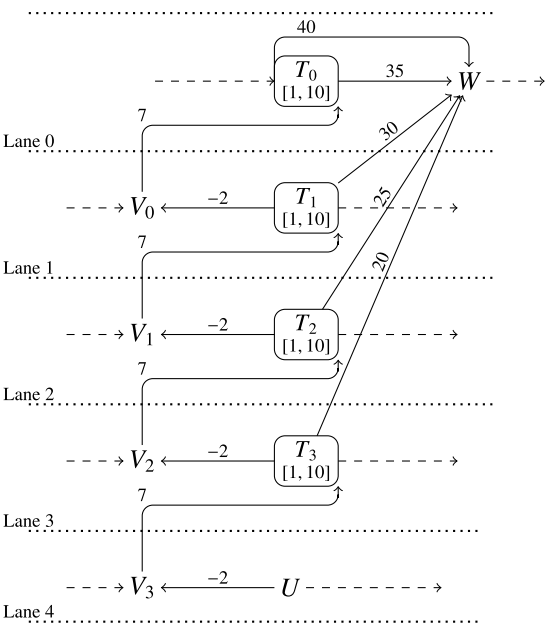


Fig. 29. Excerpt of a business plan where temporal constraints among tasks determine the quadruply-nested diamond depicted in Fig. 34a.

To assess the impact of nested diamond structures on the performance of $\texttt{minDisp}_{\text{ESTNU}}$ and $\texttt{minDisp}^{+}_{\text{ESTNU}}$, we generated a new benchmark comprising random STNU instances that each contain one copy of the quadruply-nested diamond structure depicted in Fig. 34a. In particular, the new benchmark was generated with the same parameters (see Table 8) used to generate the benchmark in the previous experiments with the addition of the four contingent links organized into the quadruply-nested diamond structure.

The STNU instances of the benchmark are a representation of random temporal business processes. In terms of temporal business processes, a quadruply-nested diamond structure like the one depicted in Fig. 34a corresponds to having four tasks, each in a proper lane, synchronized by some temporal constraints with respect to a common event W, as shown in Fig. 29. In particular, each task T_i is represented in the STNU by a contingent link $(A_i, 1, 10, C_i)$, where A_i represents the starting instant of the task and C_i the ending one, each event V_i or W is represented by a timepoint, and control-flow edges and inter-task constraints are represented by ordinary constraints.

The presence of the quadruply-nested diamond structure in each instance guarantees that the `for` loop in `genStandIns` (Algorithm 24) of the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm (Algorithm 23) must do at least five iterations: four to generate the proper stand-in edges (V_i, θ_i, W), and one to verify that no further propagations are possible, resulting in five calls to Johnson's algorithm to update the distance matrix. In contrast, $\texttt{minDisp}^{+}_{\text{ESTNU}}$ replaces `genStandIns` with `newGenStandIns` (Algorithm 29) where Johnson's algorithm is called at most twice, regardless of how deeply nested any diamond structure might be.

We executed the $\texttt{minDisp}_{\text{ESTNU}}$ and $\texttt{minDisp}^{+}_{\text{ESTNU}}$ algorithms on the dispatchable networks generated by the $\texttt{FD}_{\text{STNU}}$ algorithm. The results are presented in Fig. 30. The average execution time of $\texttt{minDisp}_{\text{ESTNU}}$ ($\texttt{FD}_{\text{STNU}}$) is significantly greater than the average execution time of $\texttt{minDisp}^{+}_{\text{ESTNU}}$ ($\texttt{FD}_{\text{STNU}}$) for all instances and, for instances having 2000 nodes, the execution time of $\texttt{minDisp}_{\text{ESTNU}}$ ($\texttt{FD}_{\text{STNU}}$) exceeds the 30-minute timeout. These results confirm that the $\texttt{minDisp}^{+}_{\text{ESTNU}}$ algorithm is significantly more efficient than the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm when input instances contain nested diamond structures, even when the number of nested diamonds is small.

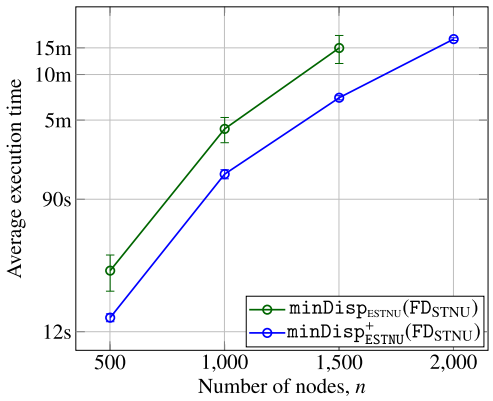
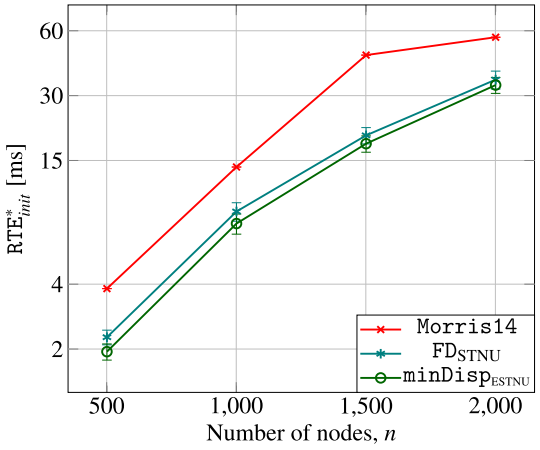
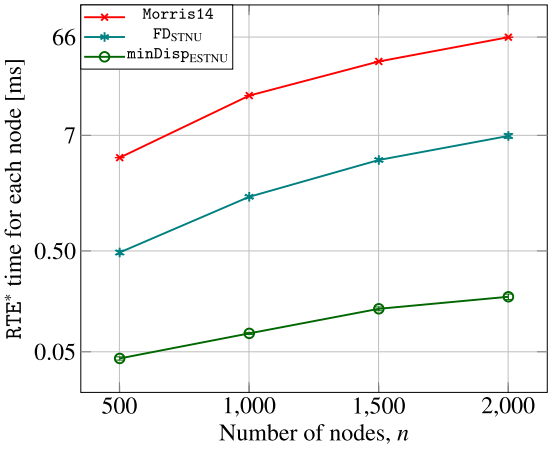


Fig. 30. $\texttt{minDisp}_{\text{ESTNU}}$ and $\texttt{minDisp}^{+}_{\text{ESTNU}}$ performance versus network size when each instance contains a quadruply-nested diamond structure (cf. Fig. 34a) that represents the business plan fragment represented in Fig. 29.



a) $\texttt{RTE}^{*}_{init}$ execution time for dispatchable ESTNUs generated by the Morris14, $\texttt{FD}_{\text{STNU}}$ and $\texttt{minDisp}_{\text{ESTNU}}$ algs. vs. number of nodes n



(b) $\texttt{RTE}^{*}$ computation time *per node* for dispatchable ESTNUs generated by Morris14, $\texttt{FD}_{\text{STNU}}$ and $\texttt{minDisp}_{\text{ESTNU}}$ vs. number of nodes

Fig. 31. Empirical results for the $\texttt{RTE}^{*}$ algorithm.

We separately checked that the improved efficiency of $\texttt{minDisp}^{+}_{\text{ESTNU}}$ becomes noticeable when there are at least triply-nested diamond structures. However, with only simple diamond structures, the execution times of the two algorithms are comparable.

8.3. *Empirical evaluation for* $\texttt{RTE}^{*}$

This section presents an empirical evaluation of the application of the $\texttt{RTE}^{*}$ real-time execution algorithm to dispatchable networks generated by the Morris14, $\texttt{FD}_{\text{STNU}}$, and $\texttt{minDisp}_{\text{ESTNU}}$ ($\texttt{FD}_{\text{STNU}}$) algorithms in the previous experiments. Figs. 31a and 31b illustrate the impact of the number of edges on the performance of the ESTNU executor $\texttt{RTE}^{*}$. For each benchmark instance, equivalent dispatchable ESTNUs were generated by Morris14, $\texttt{FD}_{\text{STNU}}$, and $\texttt{minDisp}_{\text{ESTNU}}$ ($\texttt{FD}_{\text{STNU}}$). (Recall that $\texttt{minDisp}_{\text{ESTNU}}\texttt{FD}_{\text{STNU}}$ and $\texttt{minDisp}_{\text{ESTNU}}\texttt{Morris14}$ effectively generate the same networks.) We configured $\texttt{RTE}^{*}$ to schedule timepoints using an early execution strategy for controllable timepoints and set up the environment to return the middle value for each contingent duration.

The plot in Fig. 31a shows the average execution time needed to initialize the executor on a logarithmic scale. The time depends in part on the number of edges in the network. The execution time when the input is from Morris14 is around double that when the input is from $\texttt{minDisp}_{\text{ESTNU}}$ ($\texttt{FD}_{\text{STNU}}$). In contrast, the number of edges in the Morris14 instances is around two orders of magnitude greater than the number of edges in the $\texttt{minDisp}_{\text{ESTNU}}$ ($\texttt{FD}_{\text{STNU}}$) instances. This is because the time necessary to initialize some general data structures dominates the time to initialize the data structure relative to the number of edges.

Fig. 31b shows the average time required by RTE* to schedule a single timepoint or to manage the occurrence of a contingent one. The advantage of having a minimal dispatchable network is evident. On average, the time RTE* takes to assign a value to a timepoint of a $\texttt{minDisp}_{\text{ESTNU}}$ (FD_{STNU}) instance is smaller by two orders of magnitude than the time it takes to assign a value to a timepoint of a Morris14 instance. This behavior is expected since, on average, the number of edges in the Morris14 instances is an order of magnitude greater than in the $\texttt{minDisp}_{\text{ESTNU}}$ (FD_{STNU}) instances. Consequently, RTE* must consider significantly more edges when propagating the effects of execution decisions.

9. Conclusions

This paper makes the following important theoretic and algorithmic contributions to the foundations and management of Simple Temporal Networks with Uncertainty. First, it introduces a new algorithm, called findSRNC, that finds semi-reducible negative (SRN) cycles in non-dynamically controllable (non-DC) STNUs. It is a modification of the fastest DC-checking algorithm for STNUs, called RUL2021, to accumulate path information while also rigorously addressing the compact representation of the SRN cycles it outputs. When given an overconstrained STNU, findSRNC can be used to identify constraints to relax or contingent durations to tighten. The findSRNC algorithm can also be used as a supporting process in an iterative algorithm for finding a DC STNU that well approximates a Probabilistic Simple Temporal Network [71,77,74,2].

Next, after summarizing the real-time execution algorithm, RTE*, first presented in Hunsberger and Posenato [41], the paper develops a new, rigorous theory of the *canonical form of nested diamond structures* in dispatchable ESTNUs. It then uses this theory to confirm the correctness of the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm [39] for solving the MinDispESTNU problem: computing an equivalent dispatchable ESTNU having a minimal number of edges. Then it introduces a novel algorithm, called $\texttt{minDisp}^{+}_{\text{ESTNU}}$, that solves the MinDispESTNU problem more efficiently by exploiting the structure of nested diamonds.

In so doing, the paper fills an important gap in the algorithmic and theoretic foundations of the dispatchability of Simple Temporal Networks with Uncertainty.

The paper concludes with a series of empirical evaluations of all of the above algorithms that demonstrate their effectiveness.

Epilogue. After submission but before publication of this article, Hunsberger and Posenato [43] published a new algorithm for solving the MinDispESTNU problem, called $\text{betterMinDisp}_{\text{ESTNU}}$, that runs in $O(mn + kn^2 + n^2 \log n)$ time. It generates stand-in edges as follows. For each timepoint W, it propagates *backward* along ordinary and wait edges in the ESTNU graph to compute all of the $d_*(\cdot, W)$ values. As each activation timepoint A is encountered during back-propagation, the values $d_*(A, W)$ and $d_*(C, W)$ are used to compute the value $\delta - \gamma = d_*(C, W) - d_*(A, W)$ that specifies the projection that maximizes $d_*(X, W)$ for any timepoint X that uses it. (Keep in mind that W is fixed.) The $\text{betterMinDisp}_{\text{ESTNU}}$ algorithm's proof of correctness is also based on the canonical form of nested diamond structures. Although it has better worst-case complexity than the $\texttt{minDisp}^{+}_{\text{ESTNU}}$ algorithm presented in this paper—$O(mn + n^2k + n^2 \log n)$ vs. $O(n^3 + k^2 n \log n)$—it may not necessarily be faster in practice because the $\text{betterMinDisp}_{\text{ESTNU}}$ algorithm generally works faster only in cases where there is a substantial degree of nesting of diamond structures. Since the form of such structures is quite strict, it may be that such nesting is rare in practice. Future work will focus on the degree of such nesting that appears in practical applications and then conduct a comprehensive empirical comparison of all algorithms for solving the MinDispESTNU problem.

CRediT authorship contribution statement

Luke Hunsberger: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Methodology, Investigation, Formal analysis, Conceptualization. **Roberto Posenato:** Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization.

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.

Appendix A. Formal analysis of nested diamond structures and the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm

This appendix first presents a rigorous and comprehensive analysis of the *canonical form of nested diamond structures,* which is a new concept that provides a solid theoretical foundation for understanding ESTNU-dispatchability analogous to how *shortest vee-paths* provide a theoretical foundation for understanding STN-dispatchability.[21] This foundation reveals numerous properties of nested diamonds that helped to uncover a significant oversight in a previous algorithm,

[21] The theoretical analysis of the canonical form of nested diamond structures was originally presented in a technical report [44].

`fastMinDisp`$_{\text{ESTNU}}$ [40] (cf. Footnote 20 in Section 7). That oversight makes that algorithm incomplete (i.e., it does not guarantee finding all stand-in edges associated with nested diamonds), making its $O(n^3)$ time-complexity irrelevant. The theoretical foundation also enables the proofs of several important theorems to be more rigorous and comprehensive as compared to the proof sketches in prior work [40]. The appendix then uses these results to provide a more detailed proof of the correctness of the `minDisp`$_{\text{ESTNU}}$ algorithm for solving the MinDispESTNU problem.

A.1. Preliminary concepts and theoretical results

This section recalls and elaborates some previously published results [40].

Lemma 2 *(Fixing Weak/Misleading Waits). Let (A, x, y, C) be any contingent link. Each* weak *wait, $E = (V, C{:}{-}v, A)$ where $v \leq x$, is equivalent to its ordinary replacement $\hat{E} = (V, -v, A)$, since for each value of $\omega_c \in [x, y]$, $|E|_{\omega_c} = -v = |\hat{E}|_{\omega_c}$ (i.e., they have the same length in each projection). Similarly, each* misleading *wait, $F = (U, C{:}{-}u, A)$ where $u > y$, is equivalent to its fixed version $\hat{F} = (U, C{:}{-}y, A)$, since for each $\omega_c \in [x, y]$, $|F|_{\omega_c} = -\omega_c = |\hat{F}|_{\omega_c}$.*

Proof. In each situation $\omega_c = C - A \in [x, y]$, $|E|_{\omega_c} = \max\{-v, -\omega_c\} = -v = |\hat{E}|$, since $-\omega_c \leq -x \leq -v$. Similarly, $|F|_{\omega_c} = \max\{-u, -\omega_c\} = -\omega_c = \max\{-y, -\omega_c\} = |\hat{F}|_{\omega_c}$, since $-\omega_c \geq -y > -u$. □

Lemma 3 *(Stand-in Edges associated with a contingent link). Let (A, x, y, C) be any contingent link; and let $\omega_c = C - A \in [x, y]$. Then:*

1. *the LC edge $e = (A, c{:}x, C)$ and its stand-in edge $\hat{e} = (A, y, C)$ satisfy: $|e|_{\omega_c} \leq |\hat{e}|$, with equality when $\omega_c = y$;*
2. *the UC edge $E = (C, C{:}{-}y, A)$ and its stand-in edge $\hat{E} = (C, -x, A)$ satisfy: $|E|_{\omega_c} \leq |\hat{E}|$, with equality when $\omega_c = x$; and*
3. *any regular wait edge $F = (V, C{:}{-}v, A)$ and its stand-in edge $\hat{F} = (V, -x, A)$ satisfy: $|F|_{\omega_c} \leq |\hat{F}|$, with equality when $\omega_c = x$.*

Proof. (1) $|e|_{\omega_c} = \omega_c \leq y = |\hat{e}|$, with equality when $\omega_c = y$; (2) $|E|_{\omega_c} = -\omega_c \leq -x = |\hat{E}|$, with equality when $\omega_c = x$; and (3) $|F|_{\omega_c} = \max\{-v, -\omega_c\} \leq -x = |\hat{F}|$, since $-v \leq -x$ for a regular wait edge and $-\omega_c \leq -x$. □

Lemma 4 *(VAC rule). Let $e = (A, c{:}x, C)$ be the LC edge for a contingent link (A, x, y, C) and let $E_w = (V, C{:}{-}v, A)$ be any regular wait edge labeled by C. Let $\mathcal{P}$ be the two-edge path consisting of E_w followed by e. Then $\mathcal{P}$ entails the ordinary edge $(V, y - v, C)$.*

Proof. In any situation ω, $|\mathcal{P}|_\omega = |E_w|_\omega + |e|_\omega = \max\{-v, -\omega_c\} + \omega_c = \max\{\omega_c - v, 0\} \leq \max\{y - v, 0\} = y - v = |(V, y - v, C)|$. (Since E_w is non-misleading, $y - v \geq 0$.) □

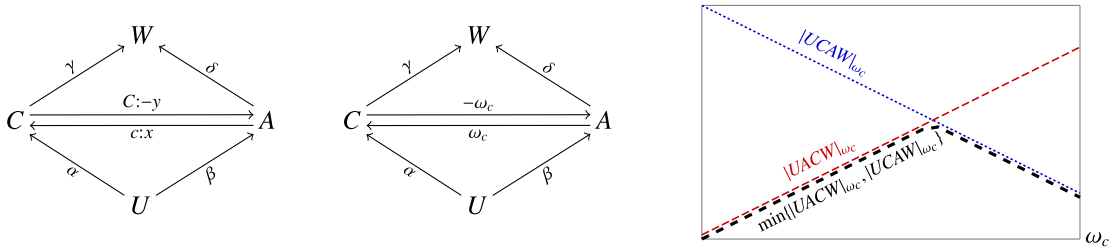
Henceforth, this section assumes that all waits are fixed and that the ESTNU $\mathcal{G}$ includes all of the stand-in edges derived from individual labeled edges and the VAC rule, as described above and as carried out by `getInitStandins` (Algorithm 25). The formal analysis below ensures that `getDiamondStandins` (Algorithm 24) correctly generates all remaining stand-in edges (i.e., those entailed by diamond structures) and correctly computes all d_* values.[22]

Definition 14 *(Labeled edge sets, contingent links, and ESTNU subgraphs).* Let $\mathcal{G}$ be any ESTNU graph; (A, x, y, C), any contingent link in $\mathcal{G}$; Λ, any set of labeled edges in $\mathcal{G}$; and U and W, any timepoints.

- An LC, UC or wait edge that is labeled by c or C is said to be *associated* with the contingent link (A, x, y, C).
- If Λ contains at least one labeled edge associated with (A, x, y, C), then (A, x, y, C) is said to be *represented* in Λ. The total number of contingent links represented in Λ is denoted by $||\Lambda||$. (Note that $||\Lambda|| = 0$ if and only if $\Lambda = \emptyset$.)
- $\Lambda^{-(A,x,y,C)}$ denotes the set of labeled edges that is the same as Λ, except that it does not include any labeled edges associated with (A, x, y, C). If the context makes the choice of contingent link clear, then we may simply write Λ^-.
- The ESTNU *subgraph induced by* Λ is the ESTNU graph $\mathcal{G}^\Lambda$ that is the same as $\mathcal{G}$ except that the only labeled edges in $\mathcal{G}^\Lambda$ are those in Λ. The timepoints and ordinary edges in $\mathcal{G}^\Lambda$ (including all stand-in edges) are the same as those in $\mathcal{G}$.
- For any situation ω for $\mathcal{G}^\Lambda$,
  - $d^\Lambda_\omega(U, W) = \min\{|\mathcal{P}|_\omega$ such that $\mathcal{P}$ is a path from U to W in $\mathcal{G}^\Lambda\}$; and
  - $d^\Lambda_*(U, W) = \max_\omega\{d^\Lambda_\omega(U, W)\}$.

  If ω is such that $d^\Lambda_\omega(U, W) = d^\Lambda_*(U, W)$, then ω is said to be *maximal* for $d^\Lambda_*(U, W)$.
- If $d_*(U, W) = d^\Lambda_*(U, W)$, then the edges in Λ are said to be *sufficient* for $d_*(U, W)$. If, in addition, no proper subset of Λ has this property, then Λ is said to be *minimally sufficient* for $d_*(U, W)$. Similarly, if $\Lambda_1 \subseteq \Lambda_2$ and $d^{\Lambda_1}_*(U, W) =$

[22] Recall that for any X and Y, the maximum length of any SVP from X to Y across all situations is denoted by $d_*(X, Y)$.



(a) A sample diamond (b) The same diamond in the situation ω_c (c) Plots of path-length functions; $\omega_c \in [x, y]$

Fig. 32. The diamond considered in Case 3 of Lemma 5.

$d_*^{\Lambda_2}(U, W)$, then the edges in Λ_1 are *sufficient* for $d_*^{\Lambda_2}(U, W)$ and if, in addition, no proper subset of Λ_1 has this property, then Λ_1 is *minimally sufficient* for $d_*^{\Lambda_2}(U, W)$.

Note. In the case where $\mathcal{G}^\Lambda = \mathcal{G}$, the definitions of d_ω^Λ and d_*^Λ reduce to d_ω and d_*, respectively. And if $\Lambda = \emptyset$ is minimally sufficient for $d_*(U, W)$, then $d_*(U, W)$ is determined solely by ordinary edges and hence equals $d(U, W)$.

Lemma 5. *Let U and W be any timepoints in a dispatchable ESTNU $\mathcal{G}$; Λ, a set of labeled edges that are minimally sufficient for $d_*(U, W)$; and (A, x, y, C), a contingent link represented in Λ. If (A, x, y, C) is the* only *contingent link represented in Λ, then for some wait edge $(V, C{:}{-}v, A)$, $\Lambda = \{(A, c{:}x, C), (V, C{:}{-}v, A)\}$, and*

$$d_*(U, W) \;=\; \mathrm{d}(U, V) + \max\{\mathrm{d}(A, W) - v, \mathrm{d}(C, W)\}.$$

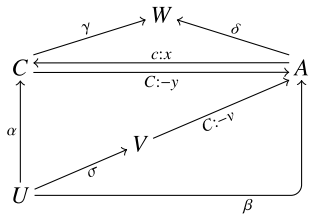
Proof. Given the premise of the lemma, $d_*(U, W) = d_*^\Lambda(U, W)$ and every projection must have a simple SVP from U to W that derives from an ESTNU path involving zero or more ordinary edges and *at least one* edge labeled by c or C. (If some projection had an SVP from U to W deriving from only ordinary edges in the ESTNU, it would contradict that Λ is minimally sufficient.)

The following cases show that Λ must include the LC edge $e = (A, c{:}x, C)$ and at least one wait edge $E_v = (V, C{:}{-}v, A)$, but not the UC edge $E = (C, C{:}{-}y, A)$.

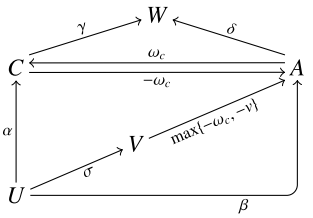
Case 1: $\Lambda = \{e\}$ (i.e., e is the only needed labeled-edge). Let ω be the situation in which $C - A = y$, the maximum duration. Since Λ is minimally sufficient for $d_*(U, W)$, there must be some ESTNU path $\mathcal{P}$ from U to W whose only labeled edge is e and for which $|\mathcal{P}|_\omega \le d_*(U, W)$. Let $\mathcal{P}^{\mathrm{si}}$ be the *ordinary* path that is the same as $\mathcal{P}$ except that e has been replaced by its stand-in edge $\hat{e} = (A, y, C)$. By Lemma 3, $|e|_\omega = y = |\hat{e}|$, whence $|\mathcal{P}^{\mathrm{si}}| = |\mathcal{P}|_\omega \le d_*(U, W)$, contradicting that Λ is minimally sufficient for $d_*(U, W)$.

Case 2: $e \notin \Lambda$ (i.e., e is not *needed).* Consider the situation ω where $C - A = x$, the minimum duration. There must be some simple ESTNU path $\mathcal{P}$ from U to W whose only labeled edge is either the UC edge or a wait edge labeled by C, and for which $|\mathcal{P}|_\omega \le d_*(U, W)$. (Since the UC edge and all wait edges labeled by C terminate at A, there can only be one such edge in any simple path.) Let $\mathcal{P}^{\mathrm{si}}$ be the *ordinary* path that is the same as $\mathcal{P}$ except that the UC or wait edge has been replaced by its stand-in edge. By Lemma 3, $|\mathcal{P}^{\mathrm{si}}| = |\mathcal{P}|_\omega \le d_*(U, W)$, contradicting that Λ is minimally sufficient for $d_*(U, W)$.

Case 3: $\Lambda = \{e, E\}$ (i.e., no wait edges are needed). For each situation ω, there must be an ESTNU path $\mathcal{P}$ from U to W that contains e or E (not both) and such that $\mathcal{P}_\omega$ is a simple shortest path in $\mathcal{G}_\omega$ for which $|\mathcal{P}|_\omega \le d_*(U, W)$. (No simple path can include both e and E.) Hence, each such path must be either *UACW* or *UCAW*, as shown in Fig. 32a, where the arrows labeled by Greek letters represent the shortest ordinary paths from the ESTNU. In any given projection, the lengths of *UACW* and *UCAW* are given by $|UACW|_{\omega_c} = \beta + \omega_c + \gamma$ and $|UCAW|_{\omega_c} = \alpha - \omega_c + \delta$, as illustrated in Fig. 32b. Note that $|UCAW|_{\omega_c}$ increases with ω_c, while $|UACW|_{\omega_c}$ decreases with ω_c, as shown in Fig. 32c. Meanwhile, the paths *UCW* and *UAW*, which comprise only ordinary edges, must satisfy: $\alpha + \gamma = |UCW| > d_*(U, W)$ and $\beta + \delta = |UAW| > d_*(U, W)$; otherwise, they would contradict that Λ is minimally sufficient. Now, since both the LC and UC edges are needed for $d_*(U, W)$, and $|UACW|_{\omega_c}$ increases with ω_c, while $|UCAW|_{\omega_c}$ decreases, it follows that the simple shortest from U to W must be *UACW* for smaller values of ω_c, and *UCAW* for larger values. Therefore, the *maximum* length of any such shortest path from U to W across any value of ω_c must occur where $|UCAW|_{\omega_c} = |UACW|_{\omega_c}$, as shown in Fig. 32c. That intersection occurs when $\omega_c = \hat{\omega}_c = \frac{1}{2}(\alpha + \delta - \beta - \gamma)$. In addition, the value of $\hat{\omega}_c$ must lie within the interval $[x, y]$, since otherwise only one of the labeled edges would be needed, contradicting that Λ is minimally sufficient. But that maximum value is $|UCAW|_{\hat{\omega}_c} = |UACW|_{\hat{\omega}_c} = \beta + \hat{\omega}_c + \gamma = \beta + \frac{\alpha+\delta-\beta-\gamma}{2} + \gamma = \frac{(\alpha+\gamma)+(\beta+\delta)}{2} = \frac{|UCW|+|UAW|}{2} > \frac{d_*(U,W)+d_*(U,W)}{2} = d_*(U, W)$, a contradiction.



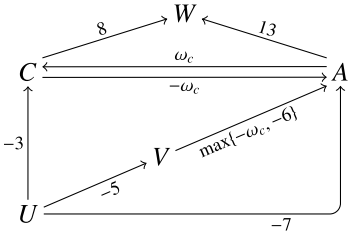
(a) ESTNU paths in a diamond structure



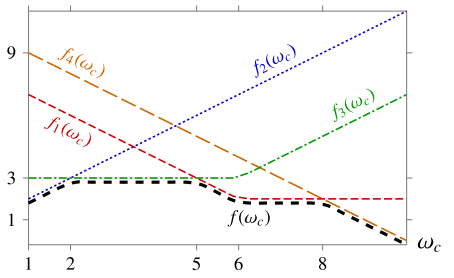
(b) The same paths in a projection, ω

$$
\begin{aligned}
f_1(\omega_c) &= |UVAW|_\omega &&= \sigma + \max\{-v, -\omega_c\} + \delta \\
f_2(\omega_c) &= |UACW|_\omega &&= \beta + \omega_c + \gamma \\
f_3(\omega_c) &= |UVACW|_\omega &&= \sigma + \max\{-v, -\omega_c\} + \omega_c + \gamma \\
f_4(\omega_c) &= |UCAW|_\omega &&= \alpha - \omega_c + \delta \\
f(\omega_c) &= \min\{f_i(\omega_c)\}
\end{aligned}
$$

(c) Path lengths as functions of $\omega_c = C - A$



(d) A sample projection with numbers



(e) Plots of path-length functions; $\omega_c \in [1, 10]$

Fig. 33. ESTNU paths and projections considered in the proof of Lemma 5.

From Cases 1–3 above, it follows that Λ must include e and at least one wait edge $E_v = (V, C{:}{-v}, A)$. For now, suppose that E_v is the only wait edge in Λ. Then, as illustrated in Fig. 33, where the arrows labeled by Greek letters represent shortest ordinary paths, each SVP from U to W in any projection $\mathcal{G}_\omega$ must, by construction, be one of the four paths, *UVAW*, *UACW*, *UVACW* or *UCAW*, whose lengths are functions of $\omega_c = C - A$, as listed in Fig. 33c. Fig. 33d shows a sample instance with numbers instead of Greek letters; and Fig. 33e plots the corresponding length functions.

Given that both e and E_v are needed for $d_*(U, W)$, the following facts are entailed.

Fact 1. $d_*(U, W) < |UCW| = \alpha + \gamma$, since $\alpha + \gamma$ is the length of the ordinary path *UCW*.
Fact 2. $d_*(U, W) < |UAW| = \beta + \delta$, since $\beta + \delta$ is the length of the ordinary path *VAW*.
Fact 3. $\beta > \sigma - v$, since otherwise $|UA|_{\omega_c} = \beta \le \sigma - v \le \sigma + \max\{-v, -\omega_c\} = |UVA|_{\omega_c}$, contradicting the need for the wait edge, $(V, C{:}{-v}, A)$.
Fact 4. $\alpha > \sigma$, since otherwise $|UCA|_{\omega_c} = \alpha - \omega_c \le \sigma - \omega_c \le \sigma + \max\{-v, -\omega_c\} = |UVA|_{\omega_c}$, contradicting the need for the wait edge $(V, C{:}{-v}, A)$.

Now, each of the length functions listed in Fig. 33c is continuous, piecewise-linear and monotone (i.e., non-decreasing or non-increasing). Thus, the *minimum* of the length functions, $f(\omega_c) = \min_{1 \le i \le 4}\{f_i(\omega_c)\}$ (black and dashed in Fig. 33e) must be continuous and piecewise linear, although not necessarily monotone. So the *maximum* value of $f(\omega_c)$ (i.e., $d_*(U, W)$) must occur at an endpoint of one of the piecewise-linear segments of $f(\omega_c)$ (i.e., where two of the length functions intersect—or where $\omega_c = x$ or $\omega_c = y$). Therefore, if $\mathcal{I}$ is the set of all such endpoints, then $d_*(U, W) = \max\{f(\omega_c) \mid \omega_c \in \mathcal{I}\}$. In the sample plot, $d_*(U, W) = 3$, which occurs for $\omega_c \in [2, 5]$.

Claims 5.1 and 5.2, below, show that f_1 and f_3 suffice to determine the value of $d_*(U, W)$.

Claim 5.1. The values of $f_2(x)$, $f_2(y)$, $f_4(x)$ and $f_4(y)$ have no bearing on the value of $d_*(U, W)$.

Proof of Claim 5.1.
- $f_4(x) = \alpha - x + \delta$ equals the length of the *ordinary* path *UCAW*, using the stand-in edge $(C, -x, A)$; so if it were equal to $d_*(U, W)$, it would contradict that Λ was minimally sufficient.
- Similarly, $f_2(y) = \beta + y + \gamma$ equals the length of the *ordinary* path *UACW*, using the stand-in edge (A, y, C); so if it were equal to $d_*(U, W)$, it would contradict that Λ was minimally sufficient.
- If $f_2(x)$ were equal to $d_*(U, W)$, and less than all other $f_i(x)$ values, then $f_2(\omega_c)$ would have to equal $f(\omega_c)$ on some interval $[x, x + \epsilon]$; but since f_2 is an increasing function, $f_2(x)$ could not be the maximum value of f on $[x, y]$.
- Similarly, if $f_4(y)$ were equal to $d_*(U, W)$, and less than all other $f_i(y)$ values, then $f_4(\omega_c)$ would have to equal $f(\omega_c)$ on some interval $[y - \epsilon, y]$, but since f_4 is a decreasing function, $f_4(y)$ could not be the maximum value of f on $[x, y]$. □

Next, let $\mathcal{I}$ be the set of values of ω_c at all points of intersection among the length functions $f_i(\omega_c)$ listed in Fig. 33c. For each i and j with $1 \le i < j \le 4$, let τ_{ij} denote the value of ω_c where $f_i(\omega_c) = f_j(\omega_c)$. For example, τ_{13} denotes the value of ω_c at which $f_1(\omega_c) = f_3(\omega_c)$ (i.e., where $|UVAW|_{\omega_c} = |UVACW|_{\omega_c}$). Intersection points that depend on the projected length of the wait edge (i.e., $\max\{-v, -\omega_c\}$) are given the superscript v (for the case where $-v \ge -\omega_c$) or ω (for $-\omega_c \ge -v$). It is easy but tedious to compute all eight possible points of intersection[23]:

$$\tau^v_{12} = \sigma + \delta - v - \beta - \gamma \qquad \tau^\omega_{12} = \frac{\sigma+\delta-\beta-\gamma}{2} \qquad \tau_{13} = \delta - \gamma \qquad \tau^v_{14} = \alpha - \sigma + v$$
$$\tau^\omega_{23} = \sigma - \beta \qquad \tau_{24} = \frac{\alpha+\delta-\beta-\gamma}{2} \qquad \tau^v_{34} = \frac{\alpha+\delta-\sigma+v-\gamma}{2} \qquad \tau^\omega_{34} = \alpha + \delta - \sigma - \gamma$$

Claim 5.2. For each $\tau \in \mathcal{I}$, $f(\tau) = \min\{f_1(\tau), f_3(\tau)\} = \min\{|UVAW|_\tau, |UVACW|_\tau\}$ (i.e., the values of $f_2(\tau)$ and $f_4(\tau)$ are irrelevant).

Proof of Claim 5.2. We show the proof for just a few of the eight cases, leaving the rest to the reader.

(τ^v_{12}) This is the case where $f_1 = f_2$ and $-v \ge -\omega_c$, yielding $\tau^v_{12} = \sigma + \delta - v - \beta - \gamma$. If $f(\tau^v_{12})$ equals $f_1(\tau^v_{12}) = f_2(\tau^v_{12})$ or $f_3(\tau^v_{12})$, then the claim holds. It only remains to show that the minimum value is *not* $f_4(\tau^v_{12}) = \alpha - (\sigma - v + \delta - \beta - \gamma) + \delta = (\alpha + \gamma) + (\beta - \sigma + v)$. Since $(\alpha + \gamma) > d_*(U, W)$, by Fact 1; and $(\beta - \sigma + v) > 0$, by Fact 3, we get that $f_4(\tau^v_{12}) > d_*(U, W)$.

(τ^ω_{12}) This is the case where $f_1 = f_2$ and $-\omega_c \ge -v$, yielding $\tau^\omega_{12} = \frac{\sigma+\delta-\beta-\gamma}{2}$. As above, it suffices to show that the minimum value is *not* $f_4(\tau^\omega_{12}) = \alpha - \tau^\omega_{12} + \delta = \frac{\alpha-\sigma}{2} + \frac{\alpha+\gamma}{2} + \frac{\beta+\delta}{2}$. Now, $(\alpha - \sigma) > 0$, by Fact 4; $(\alpha + \gamma) > d_*(U, W)$, by Fact 1; and $(\delta + \beta) > d_*(U, W)$, by Fact 2. Hence, $f_4(\tau^\omega_{12}) > \frac{\alpha+\gamma}{2} + \frac{\delta+\beta}{2} > \frac{d_*(U,W)}{2} + \frac{d_*(U,W)}{2} = d_*(U, W)$.

(τ_{13}) This is the case where $f_1 = f_3$, yielding $\tau_{13} = \delta - \gamma$. Then $f_2(\tau_{13}) = \beta + \tau_{13} + \gamma = \beta + (\delta - \gamma) + \gamma = \beta + \delta > d_*(U, W)$, by Fact 2. Similarly, $f_4(\tau_{13}) = \alpha - \tau_{13} + \delta = \alpha - (\delta - \gamma) + \delta = \alpha + \gamma > d_*(U, W)$, by Fact 1. Hence, neither $f_2(\tau_{13})$ nor $f_4(\tau_{13})$ can be the minimal value. □

Given Claims 5.1 and 5.2, the paths, *UVAW* and *UVACW*, suffice to determine the value $d_*(U, W)$, which requires computing only their *single* point of intersection: $\tau_{13} = \delta - \gamma$. At that point, $f_1(\tau_{13}) = f_1(\delta - \gamma) = \sigma + \max\{-v, -(\delta - \gamma)\} + \delta = \sigma + \max\{\delta - v, \gamma\} = \mathrm{d}(U, V) + \max\{\mathrm{d}(A, W) - v, \mathrm{d}(C, W)\}$. Hence:

$$d_*(U, W) = \mathrm{d}(U, V) + \max\{\mathrm{d}(A, W) - v, \mathrm{d}(C, W)\}$$

In the sample plot from Fig. 33e, $\delta - \gamma = 13 - 8 = 5$ and $d_*(U, W) = f_1(5) = -5 + \max\{13 - 6, 8\} = -5 + 8 = 3$.

Claim 5.3. $x < \delta - \gamma \le y$.

Proof of Claim 5.3. If $\delta - \gamma < x$, then let $\mathcal{P}_{cw}$ be the ordinary path obtained by concatenating $(C, -x, A)$ and AW, where $(C, -x, A)$ is the stand-in edge for the UC edge $(C, C{:}{-y}, A)$. Then $|\mathcal{P}_{cw}| = -x + \delta < \gamma$, since $\delta - \gamma < x$. But this contradicts that γ is the shortest path-length from C to W. Similarly, if $\delta - \gamma > y$, then let $\mathcal{P}_{aw}$ be the ordinary path obtained by concatenating (A, y, C) and CW, where (A, y, C) is the stand-in edge for the LC edge $(A, c{:}x, C)$. Then $|\mathcal{P}_{aw}| = y + \gamma < \delta$, since $\delta - \gamma > y$. But this contradicts that δ is the shortest path-length from A to W. Finally, if $\delta - \gamma = x$, then let $\mathcal{P}_{uw}$ be the ordinary path obtained by concatenating UV, $(V, -x, A)$ and AW, where $(V, -x, A)$ is the stand-in edge for $(V, C{:}{-y}, A)$. Then $|\mathcal{P}_{uw}| = \sigma - x + \delta = \sigma + \gamma$, since $\delta - \gamma = x$. Meanwhile:

$$\begin{aligned} d_*(U, W) &= \sigma + \max\{\delta - v, \gamma\} \\ &= \sigma + \max\{\gamma + x - v, \gamma\} && \text{(Since } \delta - \gamma = x\text{)} \\ &= \sigma + \gamma && \text{(Since } x < v \text{ for all regular waits)} \end{aligned}$$

Hence, $|\mathcal{P}_{uw}| = d_*(U, W)$, contradicting the need for labeled edges associated with (A, x, y, C). □

Claim 5.4. *UV* necessarily comprises only negative edges, while *CW* necessarily comprises only non-negative edges.

[23] τ^ω_{14} is undefined since $f^\omega_1(\omega_c) = f_4(\omega_c)$ holds only if $\sigma = \alpha$, which is impossible, by Fact 4. Similarly, τ^v_{23} is undefined since $f_2(\omega_c) = f^v_3(\omega_c)$ holds only if $\beta = \sigma - v$, which is impossible, by Fact 3.

Proof of Claim 5.4. First, consider the values of $f_2(\delta-\gamma)$ and $f_4(\delta-\gamma)$:

$$f_2(\delta-\gamma) = \beta + (\delta-\gamma) + \gamma = \beta + \delta > d_*(U,W).$$
$$f_4(\delta-\gamma) = \alpha - (\delta-\gamma) + \delta = \alpha + \gamma > d_*(U,W).$$

Since both of these are greater than $d_*(U,W)$, while $f_1(\delta-\gamma) = f_3(\delta-\gamma) = d_*(U,W)$, it follows that for all ω_c in some open interval around $\delta-\gamma$, $\min\{f_1(\omega_c), f_3(\omega_c)\} < \min\{f_2(\omega_c), f_4(\omega_c)\}$. In addition, since Claim 5.3 gives that $x < \delta-\gamma$, we may conclude that for some $\epsilon > 0$, the above inequality holds for all $\omega_c \in (x+\epsilon, \delta-\gamma) \subseteq (x, \delta-\gamma)$. Furthermore, for all such ω_c:

$$\begin{array}{rcll} f_3(\omega_c) & = & \sigma + \max\{-v, -\omega_c\} + \omega_c + \gamma & \\ & < & \sigma + \max\{-v, -\omega_c\} + \delta & (\text{Since } \omega_c < \delta-\gamma) \\ & = & f_1(\omega_c) & \\ & < & \min\{f_2(\omega_c), f_4(\omega_c)\} & \end{array}$$

That implies that for all such ω_c, *UVACW* must be an SVP, in which case the subpaths *UV* and *CW* must also be SVPs. Since *UV* precedes a negative edge (namely, the wait edge), and *CW* follows a non-negative edge (namely, the LC edge), the result follows. □

Note 1. Up to this point, we have assumed that Λ contained only one wait edge $E_v = (V, C{:}{-}v, A)$. Now consider the possibility of Λ containing other wait edges. For any other wait edge $E_{v'} = (V', C{:}{-}v', A)$, the value of $\delta-\gamma$ is fixed, but the value of $f(\delta-\gamma) = \sigma + \max\{\delta - v', \gamma\}$ depends on both $\sigma = d(U, V')$ and v'. Therefore, from all of the wait edges in Λ, let E_v be one that *minimizes* the value of $f(\delta-\gamma)$. Then, by the preceding analysis, for each value of ω_c, the corresponding projection must include an SVP $\mathcal{P}_v$ from U to W for which $|\mathcal{P}_v|_{\omega_c} \le f(\delta-\gamma)$. Therefore, $f(\delta-\gamma) \le d_*(U,W)$. But, by the selection of E_v, no *shorter* vee-path from U to W exists in the projection where $\omega_c = \delta-\gamma$. Therefore, $f(\delta-\gamma) = d_*(U,W)$; and since Λ is minimally sufficient for $d_*(U,W)$, it follows that Λ cannot contain more than one such wait edge.

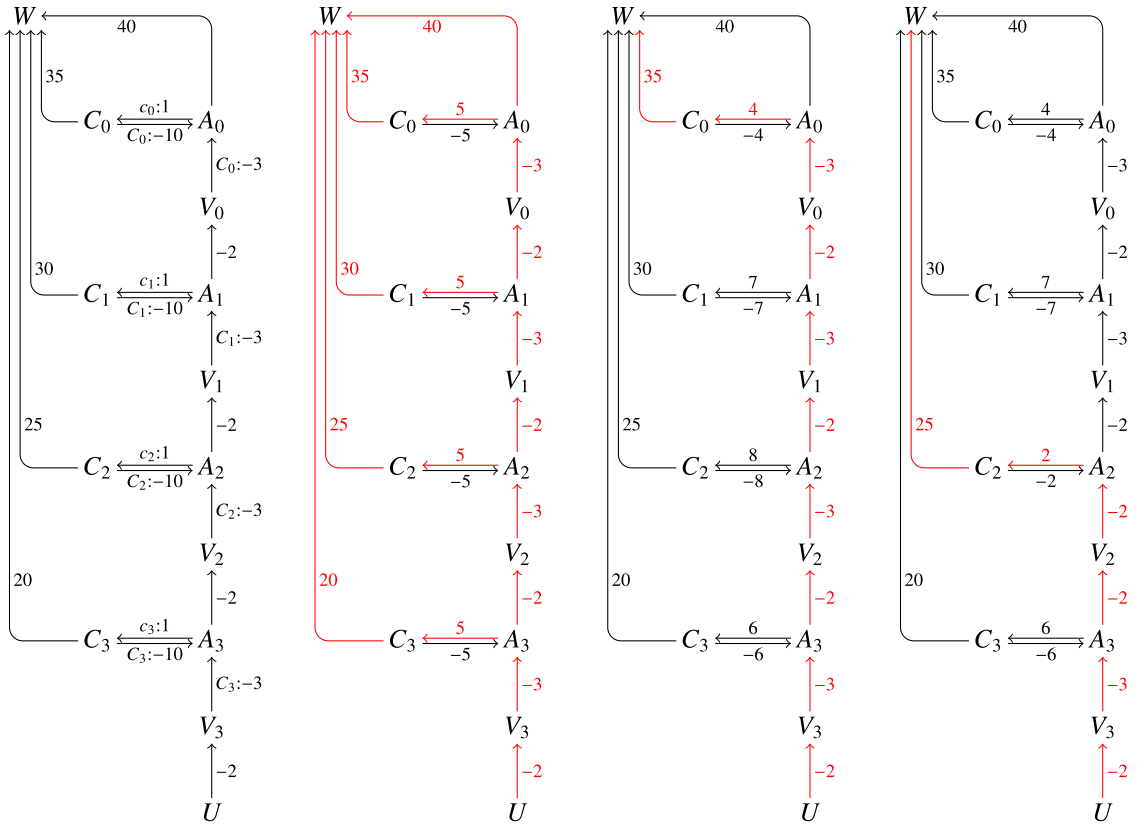
Note 2. Finally, the proof so far has assumed that $W \notin \{A, C\}$. $W \not\equiv A$ is justified since the projection of the LC edge $(A, c{:}x, C)$ in any situation cannot be in any simple SVP ending in A and hence cannot be needed by $d_*(U,A)$. In addition, since the UC edge and any wait edge labeled by C necessarily point at A, any SVP from U to A whose last edge derived from a UC or wait edge would, in the situation where $\omega_c = x$, have length $-x$ (i.e., the length of the corresponding stand-in edge), implying that the UC or wait edge was not needed. $W \not\equiv C$ is justified since the only way that a wait edge $(V, C{:}{-}v, A)$ can be needed for $d_*(U,C)$ is if, in each projection, there is an SVP that includes the concatenation of $(V, C{:}{-}v, A)$ and the LC edge $(A, c{:}x, C)$. The length of that two-edge subpath is maximized when $\omega_c = y$, where: $|VAC|_y = \max\{-v, -y\} + y = \max\{y - v, 0\} = y - v$. ($y - v > 0$, given the assumption that all misleading waits have been fixed.) But then, in that projection, *VAC* is dominated by the stand-in edge, $(V, y-v, C)$, generated by the VAC rule, which is presumed to already be in the network, which implies that $(V, C{:}{-}v, A)$ cannot be needed for $d_*(U,C)$.

Similar arguments show that $U \notin \{A, C\}$ (i.e., the contingent link (A, x, y, C) cannot be needed for $d_*(A,W)$ or $d_*(C,W)$). In particular, if $U \equiv A$, then the only labeled edge whose projection could participate in any simple SVP emanating from A would be the LC edge $(A, c{:}x, C)$. But in the situation where $\omega_c = y$, the projection of that LC edge is its stand-in edge (A, y, C), which would imply that there was an ordinary path from A to W with length at most $d_*(A,W)$, contradicting the need for edges associated with (A, x, y, C). Similarly, if $U \equiv C$, then the only labeled edge whose projection could participate in any simple SVP emanating from C would be the UC edge $(C, C{:}{-}y, A)$. But in the situation where $\omega_c = x$, the projection of that UC edge is its stand-in edge $(C, -x, A)$, which would imply that there was an ordinary path from C to W with length at most $d_*(C,W)$, contradicting the need for edges associated with (A, x, y, C). □

A.2. The canonical form of nested diamond structures

This section introduces a novel and rigorous theoretical treatment of the *canonical form of nested diamond structures*. It shows that if a value $d_*(U,W)$ depends on nested diamonds, then there is necessarily a chain of nested diamonds with certain characteristics. Fig. 34a shows a sample ESTNU graph that illustrates the canonical form of nested diamonds in the case of four contingent links. For ease of discussion, the contingent links all have the same bounds: $[1, 10]$; the wait edges all have the same value: -3; and the ordinary edges preceding the wait edges all have the same value: -2. In general, for each C_i, there is a shortest path from C_i to W that comprises one or more non-negative ordinary edges, while for each V_i, there is a shortest path from A_{i+1} to V_i that comprises zero or more negative ordinary edges. (Let $A_4 = U$.) Fig. 34b shows the *maximizing* situation/projection where each contingent link has the same duration: 5. In this situation, all of the red paths from U to W have the same length: $d_*(U,W) = 20$. In particular, letting

$$\Lambda_0 = \{(A_0, 1, 10, C_0), (V_0, C_0{:}-3, A_0)\}$$
$$\Lambda_1 = \Lambda_0 \cup \{(A_1, 1, 10, C_1), (V_1, C_1{:}-3, A_1)\}$$
$$\Lambda_2 = \Lambda_1 \cup \{(A_2, 1, 10, C_2), (V_2, C_2{:}-3, A_2)\}$$



(a) ESTNU paths

(b) Projection $\omega = (5,5,5,5)$, $d_\omega(U,W) = 20 = d_*(U,W)$

(c) Projection $\omega = (4,7,8,6)$, $d_\omega(U,W) = 19 \le d_*(U,W)$

(d) Projection $\omega = (4,7,2,6)$, $d_\omega(U,W) = 18 \le d_*(U,W)$

Fig. 34. Illustrating the canonical form of nested diamond structures.

$$\Lambda_3 = \Lambda_2 \cup \{(A_3, 1, 10, C_3), (V_3, C_3{:}-3, A_3)\}$$

observe that

$C_0 - A_0 = 5$ is maximizing for $d_*^{\Lambda_0}(A_1, W)$;
$C_1 - A_1 = C_0 - A_0 = 5$ is maximizing for $d_*^{\Lambda_1}(A_2, W)$;
$C_2 - A_2 = C_1 - A_1 = C_0 - A_0 = 5$ is maximizing for $d_*^{\Lambda_2}(A_3, W)$; and
$C_3 - A_3 = C_2 - A_2 = C_1 - A_1 = C_0 - A_0 = 5$ is maximizing for $d_*^{\Lambda_3}(U, W)$.

Similarly to the situation in Lemma 5 where the paths *UVAW* and *UVACW* have the same length in the maximizing situation where $C - A = \delta - \gamma$, for each $i \in \{0,1,2,3\}$ in Fig. 34b, $C_i - A_i = \delta_i - \gamma_i = 5$, where $\delta_i = d_*^{\Lambda_{i-1}}(A_i, W)$ and $\gamma_i = \mathrm{d}(C_i, W)$, ensures that the paths $A_{i+1}V_iA_iW$ and $A_{i+1}V_iA_iC_iW$ have the same lengths. (Let $\Lambda_{-1} = \emptyset$.)

For any situation ω, it will be shown that $d_\omega^{\Lambda_3}(U,W) \le d_*^{\Lambda_3}(U,W)$. For example, in the situation/projection where $\omega = (4,7,8,6)$, Fig. 34c shows that the SVP from U to W (shown in red) follows the chain of activation timepoints until it hits A_0, where the SVP follows the edges $(A_0, 4, C_0)$ and $(C_0, 35, W)$. In that situation, $d_\omega^{\Lambda_3}(U,W) = 19 \le 20 = d_*^{\Lambda_3}(U,W)$. Similarly, in the situation/projection where $\omega = (4,7,2,6)$, Fig. 34d shows that the SVP from U to W (shown in red) follows the chain of activation timepoints until it hits A_2, where it then follows the edges $(A_2, 2, C_2)$ and $(C_2, 25, W)$, resulting in $d_\omega^{\Lambda_3}(U,W) = 18 \le 20 = d_*^{\Lambda_3}(U,W)$. This highlights the similarity with Lemma 5 where, depending on the situation, sometimes the SVP follows *VAW*, while other times it follows *VACW*.

The theorem below addresses several challenges. First, it can be used to recursively specify the chain of contingent links from U to W that form the backbone of the structure of nested diamonds. Second, it confirms that each $d_*^{\Lambda_i}(C_i, W)$ value is realized by an ordinary path. Third, it can be used to recursively ensure (from W back to U) that any maximizing situation for each $d_*^{\Lambda_i}(A_i, W)$ can be extended to a maximizing situation for the preceding $d_*^{\Lambda_{i+1}}(A_{i+1}, W)$. Fourth, it confirms that each pair of consecutive sets of labeled edges, Λ_{i+1} and Λ_i, differ only in that Λ_i represents exactly one fewer contingent link. The end result is that each $d_*(U,W)$ value is necessarily realized by a structure of nested diamonds in canonical form.

Because it cannot be assumed in advance that the numbers of contingent links represented by successive Λ_i sets differ by exactly one, the statement of the theorem uses the following notation: for each $i \in \{j, j-1, \ldots\}$, $\Lambda_i \subseteq \Lambda_i^+$ are sets of labeled edges such that Λ_i is minimally sufficient for $d_*^{\Lambda_i^+}(A_{i+1}, W)$ (where $A_{j+1} = U$); and $\Lambda_i^- \subseteq \Lambda_i$ is the same as Λ_i except that it excludes the labeled edges for one contingent link. Then Λ_i^- becomes Λ_{i-1}^+ for the next iteration. Eventually, it is proven that $\Lambda_i^+ = \Lambda_i$ for each $i < j$.

Theorem 3. *Let $\mathcal{G}$ be any dispatchable ESTNU; and W, any timepoint in $\mathcal{G}$. For each integer $j > 0$, let $P(j)$ be the following proposition:*

Let U be any timepoint in $\mathcal{G}$, and let Λ_j and Λ_j^+ be any sets of labeled edges from $\mathcal{G}$ such that: (1) $\Lambda_j \subseteq \Lambda_j^+$; (2) Λ_j represents j contingent links; and (3) Λ_j is minimally sufficient for $d_*^{\Lambda_j^+}(U, W)$. Then there exists a contingent link $\mathrm{CL}_j = (A_j, x_j, y_j, C_j)$ and an associated wait edge $(V_j, C_j{:}{-}v_j, A_j)$ such that A_j is not *constrained to occur* before *the activation timepoint of any other contingent link represented in Λ_j, and:*

$$d_*^{\Lambda_j}(U, W) \;=\; d_*^{\Lambda_j^+}(U, W) \;=\; \mathrm{d}(U, V_j) + \max\{\delta_j - v_j, \gamma_j\},$$

where:

- *$\delta_j = d_*^{\Lambda_j^-}(A_j, W)$ and $\gamma_j = d_*^{\Lambda_j^-}(C_j, W) = \mathrm{d}(C_j, W)$ (i.e., an ordinary ESTNU path from C_j to W provides an SVP in each situation ω_j^- for $\mathcal{G}^{\Lambda_j^-}$), where:*
- *$\Lambda_j^- = \Lambda_j{}^{-\mathrm{CL}_j}$ is the same as Λ_j, except that it does not include labeled edges associated with CL_j; and*
- *as illustrated in Fig. 35a, there exists a path UV_j from U to V_j comprising only negative ordinary edges for which $|UV_j| = \mathrm{d}(U, V_j)$; and a path C_jW from C_j to W comprising only non-negative ordinary edges for which $|C_jW| = \mathrm{d}(C_j, W)$.*

Furthermore, for any situation ω_j^- for $\mathcal{G}^{\Lambda_j^-}$ that is maximal for $d_*^{\Lambda_j^-}(A_j, W)$, the situation ω_j for Λ_j that extends ω_j^- by specifying the duration $C_j - A_j = \delta_j - \gamma_j$ is necessarily maximal for $d_*^{\Lambda_j}(U, W) = d_*^{\Lambda_j^+}(U, W)$; and, as also illustrated in Fig. 35a:

- *the path $UV_jA_jC_jW$ (blue in the figure) obtained by concatenating the path UV_j, the wait edge $(V_j, C_j{:}{-}v_j, A_j)$, the LC edge $(A_j, c_j{:}{-}x_j, C_j)$, and the path C_jW satisfies $|UV_jA_jC_jW|_{\omega_j} = d_*^{\Lambda_j}(U, W) = d_*^{\Lambda_j^+}(U, W)$; and*
- *if A_jW is any path in $\mathcal{G}^{\Lambda_j^-}$ from A_j to W such that $|A_jW|_{\omega_j^-} = d_*^{\Lambda_j^-}(A_j, W)$, then the path UV_jA_jW (red in the figure) obtained by concatenating UV_j, the wait edge $(V_j, C_j{:}{-}v_j, A_j)$, and A_jW satisfies $|UV_jA_jW|_{\omega_j} = d_*^{\Lambda_j}(U, W)$.*

Then $P(j)$ holds for all $j > 0$.

Proof. By strong induction on $j = ||\Lambda_j||$, the number of contingent links represented in Λ_j. Suppose that $P(i)$ holds for all $i < j$. We must prove that $P(j)$ holds. Let U and $\Lambda_j \subseteq \Lambda_j^+$ be as in the statement of the theorem. Thus, Λ_j is minimally sufficient for $d_*^{\Lambda_j^+}(U, W)$. Let $\mathrm{CL}_j = (A_j, x_j, y_j, C_j)$ be any contingent link represented in Λ_j such that A_j is *not* constrained to occur *before* the activation timepoint for any *other* contingent link represented in Λ_j. Such a contingent link must exist since otherwise $\mathcal{G}$ would contain a negative cycle of precedence constraints and hence could not be dynamically controllable.

Let Λ_{C_j} be the subset of Λ_j that *only* includes edges associated with the contingent link CL_j. Let $e_j = (A_j, c_j{:}x_j, C_j)$ be the LC edge associated with CL_j, and let $E_j = (C_j, C_j{:}{-}y_j, A_j)$ be the UC edge. The following cases mirror the corresponding cases from Lemma 5.

Case 1: $\Lambda_{C_j} = \{e_j\}$. Let ω_j^- be any situation for $\mathcal{G}^{\Lambda_j^-}$ and $\hat{\omega}_j$ be the situation for $\mathcal{G}^{\Lambda_j}$ that is the same as ω_j^- except that it also specifies the duration $C_j - A_j = y_j$ (i.e., its maximum duration). Then there must be an ESTNU path $\mathcal{P}$ from U to W that includes e_j and whose projection in the situation $\hat{\omega}_j$ has length at most $d_*^{\Lambda_j}(U, W)$. But then modifying $\mathcal{P}$ by replacing e_j by its ordinary stand-in edge (A, y, C) yields a path $\mathcal{P}'$ that only uses edges from $\mathcal{G}^{\Lambda_j^-}$ and whose projection in any situation ω_j that is the same as ω_j^- except possibly for the value of the duration $C_j - A_j$ is at most $d_*^{\Lambda_j}(U, W)$. Since the choice of ω_j^- was arbitrary, it follows that e_j is not needed for $d_*^{\Lambda_j}(U, W)$, which is a contradiction.

Case 2: $e_j \notin \Lambda_{C_j}$. Let ω_j^- be any situation for $\mathcal{G}^{\Lambda_j^-}$ and $\hat{\omega}_j$ be the situation for $\mathcal{G}^{\Lambda_j}$ that is the same as ω_j^- except that it also specifies the duration $C_j - A_j = x_j$ (i.e., its minimum duration). Then there must be a *simple* ESTNU path $\mathcal{P}$ from U to W that includes either E_j or some wait edge $F_j = (V_j, C_j{:}{-}v_j, A_j)$ and whose projection has length at most $d_*^{\Lambda_j}(U, W)$. But modifying $\mathcal{P}$ by replacing E_j or F_j by its ordinary stand-in edge, $(C_j, -x_j, A_j)$ or $(V_j, -x_j, A_j)$, respectively, yields a

path $\mathcal{P}'$ that only uses edges from $\mathcal{G}^{\Lambda_j^-}$ and whose projection in any situation ω_j that is the same as ω_j^- except possibly for the value of the duration $C_j - A_j$ is at most $d_*^{\Lambda_j}(U, W)$. Since the choice of ω_j^- was arbitrary, it follows that the edges labeled by C_j are not needed for $d_*^{\Lambda_j}(U, W)$, which is a contradiction.

Case 3: $\Lambda_{C_j} = \{e_j, E_j\}$ *(i.e., no wait edges are needed).* Given that Λ_j is minimally sufficient for $d_*^{\Lambda_j^+}(U, W)$, and Λ_j^- does not include any labeled edges associated with CL_j, it follows that there must be some situation ω_j^- for $\mathcal{G}^{\Lambda_j^-}$ for which any SVP from U to W has length greater than $d_*^{\Lambda_j^+}(U, W)$. Next, let $\alpha_j^- = d_{\omega_j^-}^{\Lambda_j^-}(U, C)$, $\beta_j^- = d_{\omega_j^-}^{\Lambda_j^-}(U, A)$, $\gamma_j^- = d_{\omega_j^-}^{\Lambda_j^-}(C, W)$, and $\delta_j^- = d_{\omega_j^-}^{\Lambda_j^-}(A, W)$. Then consider the family Ω_j of situations over $\mathcal{G}^{\Lambda_j}$ where each $\omega_j \in \Omega_j$ is the same as ω_j^- except that it also specifies some value for $C_j - A_j$. By construction, as the value of $C_j - A_j$ varies across the situations $\omega_j \in \Omega_j$, the values of $\alpha_j^-, \beta_j^-, \gamma_j^-$ and δ_j^- are fixed. In addition, $\alpha_j^- + \gamma_j^- > d_*^{\Lambda_j^+}(U, W)$ and $\beta_j^- + \delta_j^- > d_*^{\Lambda_j^+}(U, W)$; and for each $\omega_j \in \Omega_j$, there must be an SVP from U to W that uses either the LC edge or the UC edge associated with CL_j. Hence, we are in the same circumstance as that in Case 3 of Lemma 5, which leads to a contradiction.

As a result of Cases 1–3 above, it follows that Λ_{C_j} must include the LC edge e_j and at least one wait edge labeled by C_j.

Considering $d_*^{\Lambda_j^-}(A_j, W)$. Let $\Lambda_j^- = \Lambda_j{}^{-\mathrm{CL}_j}$ be the same as Λ_j, except that it does not include any labeled edges associated with the contingent link CL_j; and let $\Lambda_{j-1} \subseteq \Lambda_j^-$ be such that Λ_{j-1} is minimally sufficient for $d_*^{\Lambda_j^-}(A_j, W)$. Since the number of contingent links represented in Λ_{j-1} is at most $j-1$, applying the inductive hypothesis to the timepoint A_j (instead of U) and $\Lambda_{j-1} \subseteq \Lambda_j^-$ (instead of $\Lambda_j \subseteq \Lambda_j^+$)[24] ensures that, as illustrated in Fig. 35b, there exists a contingent link $\mathrm{CL}_{j-1} = (A_{j-1}, x_{j-1}, y_{j-1}, C_{j-1})$ and an associated wait edge $(V_{j-1}, C_{j-1}{:}{-}v_{j-1}, A_{j-1})$ such that A_{j-1} is not constrained to occur before the activation timepoint of any other contingent link represented in Λ_{j-1}, and:

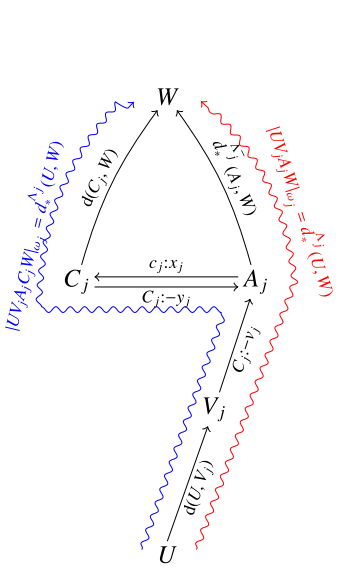
$$d_*^{\Lambda_{j-1}}(A_j, W) \;=\; d_*^{\Lambda_j^-}(A_j, W) \;=\; \mathrm{d}(A_j, V_{j-1}) + \max\{\delta_{j-1} - v_{j-1}, \gamma_{j-1}\}, \text{ where:}$$

- $\delta_{j-1} = d_*^{\Lambda_{j-1}^-}(A_{j-1}, W)$ and $\gamma_{j-1} = d_*^{\Lambda_{j-1}^-}(C_{j-1}, W) = \mathrm{d}(C_{j-1}, W)$ (i.e., an ordinary ESTNU path from C_{j-1} to W provides an SVP in each situation ω_{j-1}^- for $\mathcal{G}^{\Lambda_{j-1}^-}$), where:
- $\Lambda_{j-1}^- = \Lambda_{j-1}{}^{-\mathrm{CL}_{j-1}}$ is the same as Λ_{j-1}, except that it does not include any labeled edges associated with the contingent link CL_{j-1}; and
- there exists a path A_jV_{j-1} from A_j to V_{j-1} comprising only negative ordinary edges for which $|A_jV_{j-1}| = \mathrm{d}(A_j, V_{j-1})$, and a path $C_{j-1}W$ from C_{j-1} to W comprising only non-negative ordinary edges for which $|C_{j-1}W| = \mathrm{d}(C_{j-1}, W)$;
- any situation ω_{j-1}^- for $\mathcal{G}^{\Lambda_{j-1}^-}$ that is maximal for $d_*^{\Lambda_{j-1}^-}(A_{j-1}, W)$ can be extended to a situation ω_{j-1} for $\mathcal{G}^{\Lambda_{j-1}}$ that is maximal for $d_*^{\Lambda_{j-1}}(A_j, W) = d_*^{\Lambda_j^-}(A_j, W)$, by setting $C_{j-1} - A_{j-1} = \delta_{j-1} - \gamma_{j-1} = d_*^{\Lambda_{j-1}^-}(A_{j-1}, W) - \mathrm{d}(C_{j-1}, W)$; and, as also illustrated in Fig. 35b:
  - the path $A_jV_{j-1}A_{j-1}C_{j-1}W$ (blue in the figure), obtained by concatenating the path A_jV_{j-1}, the wait edge $(V_{j-1}, C_{j-1}{:}{-}v_{j-1}, A_{j-1})$, the LC edge $(A_{j-1}, c_{j-1}{:}{-}x_{j-1}, C_{j-1})$, and the path $C_{j-1}W$, satisfies $|A_jV_{j-1}A_{j-1}C_{j-1}W|_{\omega_{j-1}} = d_*^{\Lambda_{j-1}}(A_j, W) = d_*^{\Lambda_j^-}(A_j, W)$; and
  - if $A_{j-1}W$ is any path in $\mathcal{G}^{\Lambda_{j-1}^-}$ from A_{j-1} to W such that $|A_{j-1}W|_{\omega_{j-1}^-} = d_*^{\Lambda_{j-1}^-}(A_{j-1}, W)$, then the path $A_jV_{j-1}A_{j-1}W$ (red In the figure), obtained by concatenating A_jV_{j-1}, the wait edge $(V_{j-1}, C_{j-1}{:}{-}v_{j-1}, A_{j-1})$, and $A_{j-1}W$, satisfy $|A_jV_{j-1}A_{j-1}W|_{\omega_{j-1}} = d_*^{\Lambda_{j-1}}(A_j, W) = d_*^{\Lambda_j^-}(A_j, W)$.

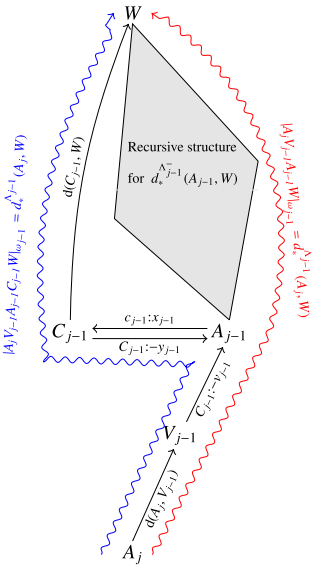
Next, proceeding recursively, as illustrated in Figs. 35c and 35d, applying the inductive hypothesis first to A_{j-1} and $\Lambda_{j-2} \subseteq \Lambda_{j-1}^-$, where Λ_{j-2} is minimally sufficient for $d_*^{\Lambda_{j-1}^-}(A_{j-1}, W)$; then to A_{j-2} and $\Lambda_{j-3} \subseteq \Lambda_{j-2}^-$, where Λ_{j-3} is minimally sufficient for $d_*^{\Lambda_{j-2}^-}(A_{j-2}, W)$; and so on, must eventually terminate, since each iteration involves a minimally sufficient set of labeled edges representing fewer contingent links. The result is a complete path structure we call the *canonical form of nested diamonds.* (Recall the canonical structures from Fig. 34.)

As illustrated in Fig. 35d, let $i = h$ be the last iteration having a *non-empty* set of labeled edges, Λ_h, that is minimally sufficient for $d_*^{\Lambda_{h+1}^-}(A_{h+1}, W)$. Let (A_h, x_h, y_h, C_h) be the corresponding contingent link, and let $\delta_h = d_*^{\Lambda_h^-}(A_h, W)$ and $\gamma_h =$

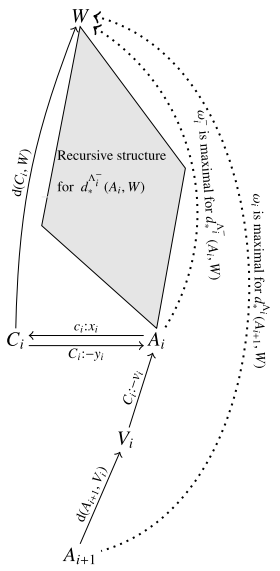
[24] For consistency of notation, it can be helpful to think of U as A_{j+1}, even though U may not be an activation timepoint for any contingent link, and Λ_j^+ as Λ_{j+1}^-, even though there is no $j+1^{\mathrm{st}}$ iteration.



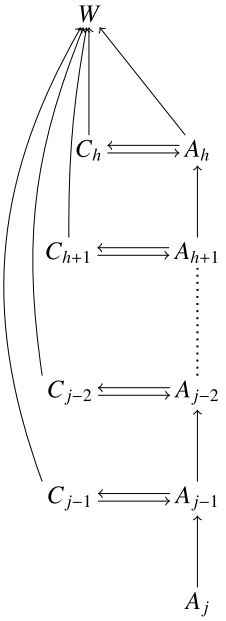
(a) Paths from the statement of Theorem 3



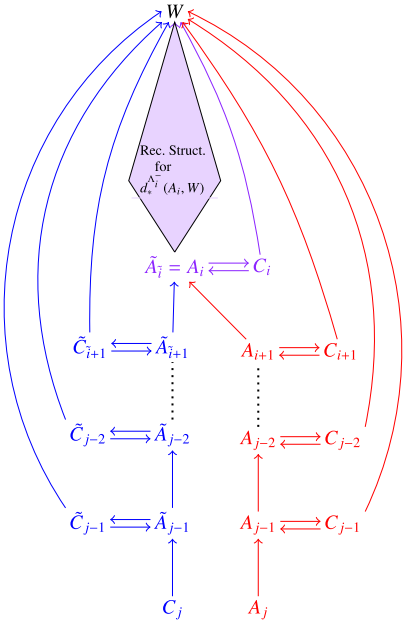
(b) Initial application of the inductive hypothesis to A_j and Λ_j^-



(c) One link in the structure



(d) Canonical structure $\mathcal{S}_{a_jw}$ from A_j to W



(e) Structures from C_j (blue) and from A_j (red) merging at $A_i = \tilde{A}_{\tilde{i}}$ (purple)

Fig. 35. Canonical structures emanating from C_j (blue) and A_j (red) analyzed in the proof of Theorem 3.

$d_*^{\Lambda_h^-}(C_h, W) = d(C_h, W)$, where Λ_h^- is the same as Λ_h, except that it does not include labeled edges from (A_h, x_h, y_h, C_h). Now, since $i = h$ is the last iteration having a non-empty set of minimally sufficient labeled edges, it follows that $\Lambda_{h-1} = \emptyset$ must be minimally sufficient for $\delta_h = d_*^{\Lambda_h^-}(A_h, W)$ and, hence, $d_*^{\Lambda_h^-}(A_h, W) = d(A_h, W)$. Therefore, the subpath from A_h to W can be presumed to contain only *ordinary* edges. Henceforth, let $\mathcal{S}_{a_jw}$ denote this *canonical structure* of paths from A_j to W, illustrated in Fig. 35d. In addition, recall that we may refer to the path from A_j to W that passes through the sequence of activation timepoints in $\mathcal{S}_{a_jw}$ as the *spine* of the canonical structure.

The sets of labeled edges resulting from this recursive application of the inductive hypothesis are summarized below where, for example, $\Lambda_{i-1} \subseteq^* \Lambda_i^-$ is used to represent not only that $\Lambda_{i-1} \subseteq \Lambda_i^-$, but also that Λ_{i-1} is minimally sufficient for $d_*^{\Lambda_i^-}(A_i, W)$:

$$\emptyset = \Lambda_{h-1} \subseteq^* \Lambda_h^- \subset \Lambda_h \subseteq^* \Lambda_{h+1}^- \subset \ldots \Lambda_{i-1} \subseteq^* \Lambda_i^- \subset \Lambda_i \subseteq^* \Lambda_{i+1}^- \ldots \Lambda_{j-2} \subseteq^* \Lambda_{j-1}^- \subset \Lambda_{j-1} \subseteq^* \Lambda_j^- \subset \Lambda_j \subseteq^* \Lambda_j^+$$

However, the following Claim implies that this chain of Λ subsets is simpler than it looks.

Claim 3.1. For each $i \in \{h, h+1, \ldots, j-1\}$, $\Lambda_{i-1} = \Lambda_i^-$ and, hence, $\Lambda_{i-1} \subseteq^* \Lambda_{i-1}$.

Proof of Claim 3.1. Suppose that $\Lambda_{j-2} \subset \Lambda_{j-1}^-$ (i.e., Λ_{j-2} is a proper subset of Λ_{j-1}^-). Then there must be some contingent link $\text{CL}^\dagger \in \Lambda_{j-1}^- \backslash \Lambda_{j-2}$, which implies that labeled edges associated with $\text{CL}^\dagger$ are not needed for $d_*^{\Lambda_{j-1}^-}(A_{j-1}, W) = \sigma_{j-2} + \max\{\delta_{j-2} - v_{j-2}, \gamma_{j-2}\}$, where $\sigma_{j-2} = d_*(A_{j-1}, V_{j-2})$ and $\gamma_{j-2} = d_*(C_{j-1}, W)$ depend only on ordinary edges. Since $\delta_{j-1} = d_*^{\Lambda_{j-1}^-}(A_{j-1}, W)$, that implies that labeled edges associated with $\text{CL}^\dagger$ are not needed for $d_*^{\Lambda_j^-}(A_j, W) = \sigma_{j-1} + \max\{\delta_{j-1} - v_{j-1}, \gamma_{j-1}\}$, where σ_{j-1} and γ_{j-1} similarly depend only on ordinary edges. But that contradicts that $\Lambda_{j-1} \subseteq^* \Lambda_j^-$. A similar analysis can be applied to each $i \in \{h, h+1, \ldots, j-1\}$. □

Since for each $i \in \{h, h+1, \ldots, j-1\}$, $\Lambda_{i-1} = \Lambda_i^-$, it follows that Λ_{i-1} represents exactly one fewer contingent link than Λ_i. The preceding chain of Λ subsets can now be more compactly written as follows, where each proper subset represents exactly one fewer contingent link:

$$\emptyset = \Lambda_{h-1} \subset \Lambda_h \subset \Lambda_{h+1} \subset \ldots \subset \Lambda_{i-1} \subset \Lambda_i \subset \ldots \Lambda_{j-2} \subset \Lambda_{j-1} \subseteq^* \Lambda_j^- \subset \Lambda_j \subseteq^* \Lambda_j^+$$

Furthermore, abbreviating contingent links (A_f, x_f, y_f, C_f) by CL_f, it follows that for each $i \in \{h, h+1, \ldots, j-1\}$, the set of contingent links represented by the subset Λ_i can be notated as $\{\text{CL}_h, \text{CL}_{h+1}, \ldots, \text{CL}_i\}$. Eventually, it will be shown that, in addition, $\Lambda_{j-1} = \Lambda_j^-$, in which case the set of contingent links represented by Λ_j will be given by $\{\text{CL}_h, \text{CL}_{h+1}, \ldots, \text{CL}_j\}$, but that has not yet been shown.

Next, we shall proceed recursively through the canonical structure $\mathcal{S}_{a_j w}$ in the opposite direction, from A_h and $\Lambda_{h-1} = \Lambda_h^-$ all the way back to A_j and $\Lambda_{j-1} \subseteq^* \Lambda_j^-$, with the goal of generating a situation ω_j^- for $\mathcal{G}^{\Lambda_j^-}$ such that not only is ω_j^- maximal for $d_*^{\Lambda_j^-}(A_j, W)$, but, in addition, for each $i \in \{h, \ldots, j-1\}$, the *restriction* of ω_j^- to the contingent links in $\mathcal{G}^{\Lambda_{i+1}^-}$ is maximal for $d_*^{\Lambda_i}(A_{i+1}, W) = d_*^{\Lambda_{i+1}^-}(A_{i+1}, W)$, where for each i, ω_j^- sets the duration $C_i - A_i = \delta_i - \gamma_i = \text{d}(C_i, W) - d_*^{\Lambda_i^-}(A_i, W)$.

We begin with $\emptyset = \Lambda_{h-1}$, which is minimally sufficient for $d_*^{\Lambda_{h-1}}(A_h, W) = d_*^{\emptyset}(A_h, W) = \text{d}(A_h, W)$. Therefore, the empty situation ω_{h-1} for $\mathcal{G}^{\Lambda_{h-1}} = \mathcal{G}^{\emptyset}$ is maximal for $d_*^{\Lambda_{h-1}}(A_h, W)$. Then, by the inductive hypothesis, ω_{h-1} can be extended to a situation ω_h for $\mathcal{G}^{\Lambda_h}$ that is maximal for $d_*^{\Lambda_h}(A_{h+1}, W)$, where ω_h sets $C_h - A_h = \delta_h - \gamma_h = \text{d}(C_h, W) - d_*^{\Lambda_{h-1}}(A_h, W)$. Continuing in this way, ω_h can be extended to a situation ω_{h+1} for $\mathcal{G}^{\Lambda_{h+1}}$ that is maximal for $d_*^{\Lambda_{h+1}}(A_{h+2}, W)$, where ω_{h+1} sets $C_{h+1} - A_{h+1} = \delta_{h+1} - \gamma_{h+1}$; and so on, until we end up with a situation ω_{j-1} that is maximal for $d_*^{\Lambda_{j-1}}(A_j, W)$, where ω_{j-1} sets $C_{j-1} - A_{j-1} = \delta_{j-1} - \gamma_{j-1}$. At this point, since $\Lambda_{j-1} \subseteq^* \Lambda_j^-$, it follows that ω_{j-1} can be extended to a situation ω_j^- for $\mathcal{G}^{\Lambda_j^-}$ by setting arbitrary durations for the contingent links, if any, that are in Λ_j^-, but not in Λ_{j-1}. In addition, by construction, not only is ω_j^- maximal for $d_*^{\Lambda_j^-}(A_j, W)$, but for each i, the restriction of ω_j^- to the contingent links in $\mathcal{G}^{\Lambda_{i+1}^-}$ is maximal for $d_*^{\Lambda_{i+1}^-}(A_{i+1}, W) = d_*^{\Lambda_i}(A_{i+1}, W)$. Furthermore, by the inductive hypothesis, since each $C_i - A_i$ equals $\delta_i - \gamma_i$, which is the value for which the path-lengths $|A_{i+1} V_i A_i C_i W|_{\omega_i}$ and $|A_{i+1} V_i A_i W|_{\omega_i}$ are equal (e.g., recall the blue and red paths from Fig. 35b), it follows that in the situation ω_j^-, the length of *every* simple path through the canonical structure $\mathcal{S}_{a_j w}$ from A_j to W equals $d_*^{\Lambda_j^-}(A_j, W)$.

Considering $d_*^{\Lambda_j^-}(C_j, W)$. Eventually, we will prove that $d_*^{\Lambda_j^-}(C_j, W) = \text{d}(C_j, W)$; however, at this point, we cannot make that assumption. For now, we can only generate the canonical path structure $\mathcal{S}_{c_j w}$ from C_j to W by recursively applying the inductive hypothesis in the same way that we did for the path structure $\mathcal{S}_{a_j w}$, above. Since there is no guarantee that the sequences of contingent links, labeled edge sets, and situations encountered along the way will be the same for $\mathcal{S}_{c_j w}$ as they were for $\mathcal{S}_{a_j w}$, we make no such assumption. Instead, we start afresh, recursively applying the inductive hypothesis to derive possibly different sequences, using different notation, as illustrated in Fig. 35e. In particular, the sequence of activation timepoints along the spine of the structure $\mathcal{S}_{c_j w}$ will be notated as $\tilde{A}_{j-1}, \tilde{A}_{j-2}, \ldots, \tilde{A}_g$; the corresponding sequence of labeled edge sets as

$$\emptyset = \lambda_{g-1} \subset \lambda_g \subset \lambda_{g+1} \subset \ldots \lambda_{\ell-1} \subset \lambda_\ell \subset \ldots \lambda_{j-2} \subset \lambda_{j-1} \subseteq^* \Lambda_j^- \subset \Lambda_j \subseteq^* \Lambda_j^+;$$

and, for each $\ell \in \{g, \ldots, j-2\}$, the situation that maximizes $d_*^{\lambda_\ell}(\tilde{A}_{\ell+1}, W)$ as $\tilde{\omega}_\ell$; and for $\ell = j-1$, the situations, $\tilde{\omega}_{j-1}$ and $\tilde{\omega}_j^-$, that maximize $d_*^{\Lambda_{j-1}}(C_j, W) = d_*^{\Lambda_j^-}(C_j, W)$.

Claim 3.2. There exists a *single* situation $\hat{\omega}_j^-$ for $\mathcal{G}_j^-$ that is maximal for *both* $d_*^{\Lambda_j^-}(A_j, W)$ and $d_*^{\Lambda_j^-}(C_j, W)$.

Proof of Claim 3.2. First, if the sequences of activation timepoints, $\{A_i\}$ and $\{\tilde{A}_\ell\}$, appearing along the respective spines of the structures $\mathcal{S}_{a_j w}$ and $\mathcal{S}_{c_j w}$ do not have any common elements, then the sets of contingent durations, $\{C_i - A_i\}$ and $\{\tilde{C}_\ell - \tilde{A}_\ell\}$, that determine the respective values of $d_*^{\Lambda_j^-}(A_j, W)$ and $d_*^{\Lambda_j^-}(C_j, W)$ are independent. Any other contingent durations specified by ω_j^- and $\tilde{\omega}_j^-$ have no impact. Therefore, a common situation $\hat{\omega}_j^-$ can be formed by merging the contingent durations, $\{C_i - A_i\}$, specified by ω_j^- with the durations, $\{\tilde{C}_\ell - \tilde{A}_\ell\}$, specified by $\tilde{\omega}_j^-$, and choosing arbitrary values for all other contingent durations, if any.

On the other hand, suppose some A_i in $\mathcal{S}_{a_j w}$ is the same as some $\tilde{A}_\ell$ in $\mathcal{S}_{c_j w}$. In case of multiple such timepoints, choose A_i to be the first activation timepoint in the spine of $\mathcal{S}_{a_j w}$ that also appears, as some $\tilde{A}_\ell$, in the spine of $\mathcal{S}_{c_j w}$, as illustrated in Fig. 35e. Then let $\Lambda_{i-1} \subseteq^* \Lambda_i^- \subset \Lambda_i$ and $\lambda_{\ell-1} \subseteq^* \lambda_\ell^- \subset \lambda_\ell$ be the corresponding sets of labeled edges, where Λ_{i-1} is minimally sufficient for $d_*^{\Lambda_i^-}(A_i, W) = d_*^{\Lambda_{i-1}}(A_i, W)$ and $\lambda_{\ell-1}$ is minimally sufficient for $d_*^{\lambda_\ell^-}(A_i, W) = d_*^{\lambda_{\ell-1}}(A_i, W)$. (By Claim 3.1, if $i \leq j-1$, then $\Lambda_{i-1} = \Lambda_i^-$; and if $\ell \leq j-1$, then $\lambda_{\ell-1} = \lambda_\ell$, but these facts are not relevant here.) Now, if $\Lambda_{i-1} = \lambda_{\ell-1}$, then no loss of generality follows from assuming that the substructures of $\mathcal{S}_{a_j w}$ and $\mathcal{S}_{c_j w}$ from A_i onward are the same, in which case the situation $\hat{\omega}_j^-$ can be obtained by merging the situations ω_j^- and $\tilde{\omega}_j^-$, as described above since, by the choice of $A_i = \tilde{A}_\ell$, there are no contingent links appearing in *both* $\mathcal{S}_{a_j w}$ and $\mathcal{S}_{c_j w}$ *before* $A_i = \tilde{A}_\ell$.

Otherwise, suppose that $\Lambda_{i-1} \neq \lambda_{\ell-1}$. Next, recall that the negative ordinary and wait edges comprising the spine of a canonical structure imposes a strict order on the participating activation timepoints. Therefore, it follows that no activation timepoints appearing *before* A_i in $\mathcal{S}_{a_j w}$ or *before* $\tilde{A}_\ell = A_i$ in $\mathcal{S}_{c_j w}$ can appear *after* A_i in $\mathcal{S}_{a_j w}$ or *after* $\tilde{A}_\ell = A_i$ in $\mathcal{S}_{c_j w}$. In other words, the contingent links represented in Λ_{i-1} and $\lambda_{\ell-1}$ are distinct from the contingent links preceding $A_i = \tilde{A}_\ell$ in either $\mathcal{S}_{a_j w}$ or $\mathcal{S}_{c_j w}$.

Now, it may happen that $d_*^{\Lambda_{i-1}}(A_i, W) \neq d_*^{\lambda_{\ell-1}}(A_i, W)$ (i.e., $\delta_i \neq \tilde{\delta}_\ell$), since either or both of these values may be greater than $d_*(A_i, W)$. To see this, recall that $\delta_{i+1} = d_*^{\Lambda_i}(A_{i+1}, W) = \sigma_i + \max\{\delta_i - v_i, \gamma_i\}$. In the case where $\gamma_i \geq \delta_i - v_i$, decreasing the value of δ_i will not change the value of δ_{i+1}. In other words, it is possible that $\delta_i > d_*(A_i, W)$, while still being sufficient to determine the value of δ_{i+1}. Similarly, it is possible that $\tilde{\delta}_\ell > d_*(A_i, W)$, while still being sufficient to determine the value of $\delta_{\ell+1}$.

Case 1: $\delta_i < \tilde{\delta}_\ell$. In this case, define the common situation $\hat{\omega}_j^-$ by first setting the duration of each contingent link CL_f that appears in $\mathcal{S}_{a_j w}$ to its value in ω_j^-. Next, set the durations for the contingent links that appear *before* $\tilde{A}_\ell = A_i$ in $\mathcal{S}_{c_j w}$ as follows:

- Let $\tilde{\delta}'_{\ell+1} = \tilde{\sigma}_\ell + \max\{\delta_i - \tilde{v}_\ell, \tilde{\gamma}_\ell\} \leq \tilde{\delta}_{\ell+1} = d_*^{\lambda_{\ell+1}^-}(\tilde{A}_{\ell+1}, W)$, since $\delta_i < \tilde{\delta}_\ell$.
- Then set $\tilde{C}_{\ell+1} - \tilde{A}_{\ell+1} = \tilde{\delta}'_{\ell+1} - \tilde{\gamma}_{\ell+1}$.
- Let $\tilde{\delta}'_{\ell+2} = \tilde{\sigma}_{\ell+1} + \max\{\tilde{\delta}'_{\ell+1} - \tilde{v}_{\ell+1}, \tilde{\gamma}_{\ell+1}\} \leq \tilde{\delta}_{\ell+2} = d_*^{\lambda_{\ell+2}^-}(\tilde{A}_{\ell+2}, W)$, since $\tilde{\delta}'_{\ell+1} \leq \tilde{\delta}_{\ell+1}$.
- Then set $\tilde{C}_{\ell+2} - \tilde{A}_{\ell+2} = \tilde{\delta}'_{\ell+2} - \tilde{\gamma}_{\ell+2}$.

  ...
- Let $\tilde{\delta}'_{j-1} = \tilde{\sigma}_{j-2} + \max\{\tilde{\delta}'_{j-2} - \tilde{v}_{j-2}, \tilde{\gamma}_{j-2}\} \leq \tilde{\delta}_{j-1} = d_*^{\lambda_{j-1}^-}(\tilde{A}_{j-1}, W)$, since $\tilde{\delta}'_{j-2} \leq \tilde{\delta}_{j-2}$.
- Then set $\tilde{C}_{j-1} - \tilde{A}_{j-1} = \tilde{\delta}'_{j-1} - \tilde{\gamma}_{j-1}$.
- Let $\tilde{\delta}'_j = \tilde{\sigma}_{j-1} + \max\{\tilde{\delta}'_{j-1} - \tilde{v}_{j-1}, \tilde{\gamma}_{j-1}\} \leq \tilde{\delta}_j = d_*^{\lambda_j^-}(C_j, W)$, since $\tilde{\delta}'_{j-1} \leq \tilde{\delta}_{j-1}$.

Finally, set all other contingent durations, if any, to arbitrary values.

Now, the setting of durations of the contingent links that appear in $\mathcal{S}_{c_j w}$ before $\tilde{A}_\ell = A_i$ is done using the corresponding $\tilde{\delta}'_p$ values which are less than or equal to the corresponding $\tilde{\delta}_p$ values. At each point along the chain, the duration $\tilde{C}_p - \tilde{A}_p$ is set to the value $\tilde{\delta}'_p - \tilde{\gamma}_p$ which, according to the proof of Lemma 5, maximizes the length of the shortest vee-path from A_{p+1} to W across all relevant projections. Since additional labeled edges are employed in the common substructure following $\tilde{A}_\ell = A_i$, that maximum value may decrease. However, the final value $\tilde{\delta}'_j$ *cannot* be less than $d_*^{\lambda_j^-}(C_j, W)$, since it would contradict the value determined by the canonical structure $\mathcal{S}_{c_j w}$. As a result, the situation $\hat{\omega}_j^-$ is necessarily maximal for both $d_*^{\Lambda_j^-}(A_j, W)$ and $d_*^{\Lambda_j^-}(C_j, W)$.

Case 2: $\delta_i < \tilde{\delta}_\ell$. Handled similarly. □

Considering $d_*^{\Lambda_j^-}(U,W)$. Although the ultimate goal is to compute $d_*^{\Lambda_j}(U,W)$, for now we consider $d_*^{\Lambda_j^-}(U,W)$: that is, the maximum length in any situation/projection of an SVP from U to W that derives from paths in $\mathcal{G}^{\Lambda_j^-}$, which *exclude* labeled edges associated with $\mathrm{CL}_j = (A_j, x_j, y_j, C_j)$. As was the case for the path structures, $\mathcal{S}_{a_jw}$ and $\mathcal{S}_{c_jw}$, seen above, recursive application of the inductive hypothesis can also be used to generate a canonical structure $\mathcal{S}_{uw}$ from U to W relevant for $d_*^{\Lambda_j^-}(U,W)$. In addition, the same kind of analysis that was done in the proof of Claim 3.2 ensures that there must be a *single* situation ω_j^- that is simultaneously maximal for the *three* values, $d_*^{\Lambda_j^-}(U,W), d_*^{\Lambda_j^-}(A_j,W)$ and $d_*^{\Lambda_j^-}(C_j,W)$, as follows. The analysis is complicated by the fact that the three structures may intersect in a variety of ways.

Claim 3.3. Given the three canonical structures, $\mathcal{S}_{a_jw}, \mathcal{S}_{c_jw}$ and $\mathcal{S}_{uw}$, there is a single situation $\breve{\omega}_j^-$ for $\mathcal{G}_j^-$ that is simultaneously maximal for the *three* values, $d_*^{\Lambda_j^-}(U,W), d_*^{\Lambda_j^-}(A_j,W)$ and $d_*^{\Lambda_j^-}(C_j,W)$.

Proof of Claim 3.3. First, let $\mathcal{A}$ denote the set of activation timepoints appearing in the spines of *any* of the three canonical structures, $\mathcal{S}_{a_jw}, \mathcal{S}_{c_jw}$ or $\mathcal{S}_{uw}$, except that $\mathcal{A}$ should not include A_j, C_j or U. Some timepoints in $\mathcal{A}$ may participate in only one structure, others may participate in two of the three structures, and still others may participate in all three structures. Next, note that the negative ordinary and wait edges along the spines of the three structures ensures that the timepoints in $\mathcal{A}$ are subject to a strict partial order, where $A_p \succ A_q$ if A_p occurs *after* A_q in at least one of the structures (i.e., A_p is closer to W than A_q). This strict partial order determines a directed acyclic graph since, otherwise, there would be a negative cycle in the OW-graph of ordinary and wait edges, contradicting the dispatchability of $\mathcal{G}$. Therefore, the timepoints in $\mathcal{A}$ can be put into a *topological sort* [19] that respects the strict partial order, perhaps notated as $A_{i_1} \succ A_{i_2} \succ A_{i_3} \succ \ldots A_{i_k}$.

Next, we will visit the timepoints in $\mathcal{A}$ in the order of that topological sort. For each activation timepoint $A \in \mathcal{A}$, we compute two values: (1) δ_A, an upper bound on $d_*(A,W)$; and (2) $C - A = \delta_A - d(C,W)$, where C is the contingent timepoint associated with A. These values are computed for each A as follows. First, A may have up to three direct successors in the strict partial order, at most one for each canonical structure. If A has no direct successors, then it is the final activation timepoint in one or more structures; hence, the value of δ_A is determined by ordinary edges: $\delta_A = \mathrm{d}(A,W)$. Otherwise, for each structure $\mathcal{S} \in \{\mathcal{S}_{a_jw}, \mathcal{S}_{c_jw}, \mathcal{S}_{uw}\}$, if that structure has an activation timepoint B that is a direct successor of A (i.e., $B \succ A$), then, as illustrated in Fig. 36, let $\delta(A,B,\mathcal{S}) = \sigma_B + \max\{\delta_B - v_B, \gamma_B\}$, where the value of δ_B was computed when B was processed, which necessarily happens before visiting A; and where $\sigma_B = d(A,V_B)$, $-v_B$ is the length of the wait edge $(V_B, C_B{:}{-}v_B, B)$, and $\gamma_B = d(C_B,W)$ are the relevant values from the canonical structure $\mathcal{S}$. If B is a direct successor of A in more than one canonical structure, then multiple $\delta(A,B,\mathcal{S})$ values will be computed for the same B. Of course, in one structure, the direct predecessor B' of A may be different from B. Regardless of how many $\delta(A,B,\mathcal{S})$ are computed, the value of δ_A is then given by $\min\{\delta(A,B,\mathcal{S}) \mid B$ is a direct successor of A in $\mathcal{S}\}$. Afterward, the duration $C - A$ is set to $\delta_A - \mathrm{d}(C,W)$. Note that this duration is chosen to maximize the value of the length of the shortest vee-path from A to W through any of the three canonical structures. As shown in Fig. 33 and the proof of Lemma 5, the duration $C_B - B = \delta_B - \gamma_B = \delta_B - d(C_B,W)$ is chosen to maximize the length of the shortest vee-path across all projections.

The terminal computations are done for A_j, C_j and U in the same way, except that each has exactly one direct successor. In particular, following earlier notation, the successor of A_j in $\mathcal{S}_{a_jw}$ is A_{j-1} and $\delta_{A_j} = \sigma_j + \max\{\delta_{A_{j-1}} - v_{j-1}, \gamma_{j-1}\}$, where $\delta_{A_{j-1}}$ was computed previously when A_{j-1} was encountered in the topological sort. Similarly, the successor of C_j in $\mathcal{S}_{c_jw}$ is $\tilde{A}_{j-1}$ and $\delta_{C_j} = \tilde{\sigma}_j + \max\{\delta_{\tilde{A}_{j-1}} - \tilde{v}_{j-1}, \tilde{\gamma}_{j-1}\}$, where $\delta_{\tilde{A}_{j-1}}$ was computed previously when $\tilde{A}_{j-1}$ was encountered in the topological sort. Finally, $\delta_U = \dot{\sigma}_j + \max\{\delta_{\dot{A}_{j-1}} - \dot{v}_{j-1}, \dot{\gamma}_{j-1}\}$, where $\delta_{\dot{A}_{j-1}}$ was computed previously when $\dot{A}_{j-1}$, the direct successor of U in $\mathcal{S}_{uw}$ was encountered.

To see why $\breve{\omega}_j^-$ is maximal for all three of the values, $d_*^{\Lambda_j^-}(U,W), d_*^{\Lambda_j^-}(A_j,W)$ and $d_*^{\Lambda_j^-}(C_j,W)$, recall from the proof of Claim 3.2 that the δ values determined by any given canonical form do not necessarily provide the shortest path lengths entailed by the network. Instead, they need only be short enough to determine the relevant $d_*(\_,W)$ values (i.e., they need only be sufficient). For example, in the structure $\mathcal{S}_{a_jw}$, the value $\delta_i = d_*^{\Lambda_i^-}(A_i,W)$ may be larger than the value $d_*(A_i,W)$. This can happen when the δ_i value does not contribute to the value of $\delta_{i+1} = \sigma_i + \max\{\delta_i - v_i, \gamma_i\}$ because $\max\{\delta_i - v_i, \gamma_i\} = \gamma_i$. In this case, setting δ_i to the value $d_*(A_i,W)$ has no effect. The δ_A values computed above effectively replace existing δ values with possibly lower values to ensure that each structure provides the desired value (i.e., $d_*^{\Lambda_j^-}(U,W), d_*^{\Lambda_j^-}(A_j,W)$ or $d_*^{\Lambda_j^-}(C_j,W)$). Since each δ_A value is the minimum of existing δ values, each δ_A cannot be lower than the corresponding $d_*(A,W)$ value. □

From here on out, the three canonical structures, $\mathcal{S}_{a_jw}$, $\mathcal{S}_{c_jw}$ and $\mathcal{S}_{uw}$, shall remain fixed; and ω_j^- shall denote the fixed situation that is simultaneously maximal for $d_*^{\Lambda_j^-}(A_j,W), d_*^{\Lambda_j^-}(C_j,W)$ and $d_*^{\Lambda_j^-}(U,W)$.

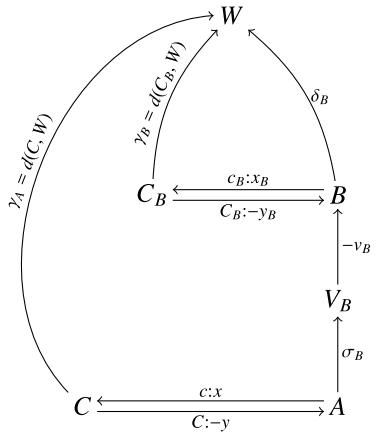
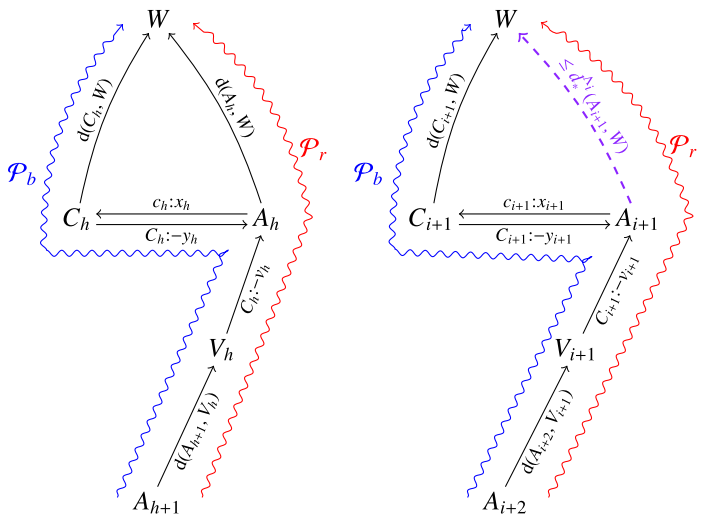


Fig. 36. Computing $\delta(A, B, \mathcal{S}) = \sigma_B + \max\{\delta_B, -v_B, \gamma_B\}$ according to Claim 3.3.



(a) Base Case of Claim 3.4 (b) Recursive Case of Claim 3.4

Fig. 37. Paths analyzed in the Proof of Claim 3.4.

The following claim ensures that in *every* situation, there is *some* path from A_j to W in the canonical structure $\mathcal{S}_{a_j w}$ whose length is at most $d^{\Lambda_j^-}_*(A_j, W)$. This, together with the fact that in the situation ω_j^- the length of every simple path through the structure $\mathcal{S}_{a_j w}$ equals $d^{\Lambda_j^-}_*(A_j, W)$, ensures that the paths constituting $\mathcal{S}_{a_j w}$ are sufficient to determine the value of $d^{\Lambda_j^-}_*(A_j, W)$.

Claim 3.4. For every situation $\hat{\omega}_j^-$ for $\mathcal{G}^{\Lambda_j^-}$ there is a simple path $A_j W$ from A_j to W that employs only edges that appear within the canonical structure $\mathcal{S}_{a_j w}$ and whose projection satisfies $|A_j W|_{\hat{\omega}_j^-} \leq d^{\Lambda_j^-}_*(A_j, W)$.

Proof of Claim 3.4. By induction. For each $i \in \{h, h+1, \ldots, j-1\}$, let $Q(i)$ be the proposition that in any situation $\hat{\omega}_i$ for the contingent links represented in Λ_i, there is a simple path $\mathcal{P}$ from A_{h+1} to W that employs only edges that appear within $\mathcal{S}_{a_j w}$ and whose projection satisfies $|\mathcal{P}|_{\hat{\omega}_i} \leq d_*^{\Lambda_i}(A_{i+1}, W)$.
Base Case: $i = h$. There are only two simple paths from A_{h+1} to W in $\mathcal{S}_{a_j w}$: $\mathcal{P}_b$ and $\mathcal{P}_r$, shown in blue and red, respectively, in Fig. 37a. The first, $\mathcal{P}_b$, goes from A_{h+1} to V_h to A_h to C_h to W; the second, $\mathcal{P}_r$, goes from A_{h+1} to V_h to A_h to W.
Let $\tau_h = \delta_h - \gamma_h = \mathrm{d}(A_h, W) - \mathrm{d}(C_h, W)$; and let $\sigma_h = \mathrm{d}(A_{h+1}, V_h)$. Now, $\hat{\omega}_h$ only specifies a single duration, namely, $C_h - A_h$. For any $\hat{\omega}_h \leq \tau_h$:

$$\begin{array}{rcll} |\mathcal{P}_b|_{\hat{\omega}_h} & = & \sigma_h + \max\{-v_h, -\hat{\omega}_h\} + \hat{\omega}_h + \gamma_h & \\ & = & \sigma_h + \max\{\hat{\omega}_h + \gamma_h - v_h, \gamma_h\} & \\ & \leq & \sigma_h + \max\{\tau_h + \gamma_h - v_h, \gamma_h\} & (\text{Since } \hat{\omega}_h \leq \tau_h) \\ & = & \sigma_h + \max\{\delta_h - v_h, \gamma_h\} & (\text{Since } \tau_h = \delta_h - \gamma_h) \end{array}$$

Meanwhile, for any $\hat{\omega}_h \geq \tau_h$:

$$\begin{array}{rcll} |\mathcal{P}_r|_{\hat{\omega}_h} & = & \sigma_h + \max\{-v_h, -\hat{\omega}_h\} + \delta_h & \\ & = & \sigma_h + \max\{\delta_h - v_h, \delta_h - \hat{\omega}_h\} & \\ & \leq & \sigma_h + \max\{\delta_h - v_h, \delta_h - \tau_h\} & (\text{Since } \hat{\omega}_h \geq \tau_h) \\ & = & \sigma_h + \max\{\delta_h - v_h, \gamma_h\} & (\text{Since } \tau_h = \delta_h - \gamma_h) \end{array}$$

As seen previously, when $\hat{\omega}_h = \tau_h$, $|\mathcal{P}_r|_{\hat{\omega}_h} = |\mathcal{P}_b|_{\hat{\omega}_h} = d_*^{\Lambda_h}(A_{h+1}, W) = \sigma_h + \max\{\delta_h - v_h, \gamma_h\}$. Therefore, for any $\hat{\omega}_h$, one of the two paths has length at most $d_*^{\Lambda_h}(A_{h+1}, W)$.
Recursive Case. Assume that $Q(i)$ holds for some $i \leq j-1$. We must show that $Q(i+1)$ holds (i.e., that in any situation $\hat{\omega}_{i+1}$ for the contingent links represented in Λ_{i+1}, there is a simple path $\mathcal{P}$ from A_{i+2} to W that employs only edges that appear within $\mathcal{S}_{a_j w}$ and for which $|\mathcal{P}|_{\hat{\omega}_{i+1}} \leq d_*^{\Lambda_{i+1}}(A_{i+2}, W)$). Let $\hat{\omega}_{i+1}$ be any situation for the contingent links represented in Λ_{i+1}. Then by $Q(i)$, the restriction of this situation to contingent links represented in Λ_i yields a path from A_{i+1} to W, shown as purple and dashed in Fig. 37b, whose length is at most $d_*^{\Lambda_i}(A_{i+1}, W)$. Let $\mathcal{P}_b$ and $\mathcal{P}_r$ be the paths shown in blue and red, respectively, in Fig. 37b, where $\mathcal{P}_b$ goes from A_{i+2} to V_{i+1} to A_{i+1} to C_{i+1} to W, while $\mathcal{P}_r$ goes from A_{i+2} to V_{i+1} to A_{i+1} to W; and let $\tau_{i+1} = \delta_{i+1} - \gamma_{i+1} = d_*^{\Lambda_i}(A_{i+1}, W) - \mathrm{d}(C_{i+1}, W)$. Let τ be the value of $C_{i+1} - A_{i+1}$ specified by $\hat{\omega}_{i+1}$. Similarly to the derivations in the base case, we get that if $\tau \leq \tau_{i+1}$, then $|\mathcal{P}_b|_{\hat{\omega}_{i+1}} \leq d_*^{\Lambda_{i+1}}(A_{i+2}, W)$; and if $\tau \geq \tau_{i+1}$, then $|\mathcal{P}_r|_{\hat{\omega}_{i+1}} \leq d_*^{\Lambda_{i+1}}(A_{i+2}, W)$. The only difference in the derivation is that the length of the purple dashed path from A_{i+1} to W is *at most* $d_*^{\Lambda_i}(A_{i+1}, W) = \delta_{i+1}$, but that does not affect the conclusion. □

Corollary 1. *Let $\hat{\omega}_j^-$ be any situation for $\mathcal{G}^{\Lambda_j^-}$; and let $C_j W$ be any simple path from C_j to W that employs only edges that appear within the canonical structure $\mathcal{S}_{c_j w}$. Then $|C_j W|_{\hat{\omega}_j^-} \leq d_*^{\Lambda_j^-}(C_j, W)$. Similarly, if UW is any simple path from U to W that employs only edges appearing within $\mathcal{S}_{uw}$, then $|UW|_{\hat{\omega}_j^-} \leq d_*^{\Lambda_j^-}(U, W)$.*

Claim 3.5. $d_*^{\Lambda_j^-}(U, W) > d_*^{\Lambda_j}(U, W)$.

Proof of Claim 3.5. Suppose that $d_*^{\Lambda_j^-}(U, W) < d_*^{\Lambda_j}(U, W)$, where the former excludes paths involving labeled edges associated with CL_j, while the latter allows them. But then, by Corollary 1, for every situation there is an ESTNU path from U to W using only edges from $\mathcal{S}_{uw}$ whose projection has length at most $d_*^{\Lambda_j^-}(U, W) < d_*^{\Lambda_j}(U, W) = d_*^{\Lambda_j^+}(U, W)$. Therefore, the maximum length of any SVP from U to W in any situation is less than $d_*^{\Lambda_j^+}(U, W)$, which is a contradiction. On the other hand, if $d_*^{\Lambda_j^-}(U, W) = d_*^{\Lambda_j}(U, W)$, then $\Lambda_j^- \subset \Lambda_j$ is sufficient for $d_*^{\Lambda_j^+}(U, W)$, contradicting that Λ_j is minimally sufficient for $d_*^{\Lambda_j^+}(U, W)$. Therefore, it must be that $d_*^{\Lambda_j^-}(U, W) > d_*^{\Lambda_j}(U, W)$. □

Now we are finally ready to construct the canonical structure from U to W, to be called $\mathcal{S}_{uw}^{\Lambda_j}$, that will have all of the properties listed in the conclusion of the theorem. Starting with the contingent link $\mathrm{CL}_j = (A_j, x_j, y_j, C_j)$, attach the structures $\mathcal{S}_{c_j w}$ and $\mathcal{S}_{a_j w}$ to the timepoints C_j and A_j, respectively, as illustrated in Fig. 38a. Next, for each wait edge $(V_j, C_j{:}{-}v_j, A_j)$ for which there is an ordinary path $\mathcal{P}_{uv_j}$ from U to V_j, attach $\mathcal{P}_{uv_j}$ and the wait edge to the timepoint A_j, as shown in the figure. At this point, we don't know how many wait edges will be needed; however, it will soon be shown that only one is needed.

Claim 3.6. In any situation ω_j for $\mathcal{G}^{\Lambda_j}$ there is a simple path $\mathcal{P}$ whose edges are drawn solely from $\mathcal{S}_{uw}^{\Lambda_j}$ and whose projection satisfies $|\mathcal{P}|_{\omega_j} \le d_*^{\Lambda_j}(U, W)$.

Proof of Claim 3.6. First, recall the situation ω_j^-, which is maximal for $d_*^{\Lambda_j^-}(U, W)$. Since, by Claim 3.5, $d_*^{\Lambda_j^-}(U, W) > d_*^{\Lambda_j}(U, W)$, it follows that in any situation $\hat{\omega}_j$ that extends ω_j^- by also specifying a duration for $C_j - A_j$, any ESTNU path in $\mathcal{G}^{\Lambda_j}$ whose projection in $\hat{\omega}_j$ is an SVP from U to W must include one or more labeled edges associated with the contingent link $\mathrm{CL}_j = (A_j, x_j, y_j, C_j)$, as illustrated in Fig. 38b, where: $\alpha_j = d_{\omega_j^-}^{\Lambda_j^-}(U, C_j)$, $\beta_j = d_{\omega_j^-}^{\Lambda_j^-}(U, A_j)$, $\sigma_j = d_{\omega_j^-}^{\Lambda_j^-}(U, V_j)$, $\gamma_j = d_{\omega_j^-}^{\Lambda_j^-}(C_j, W)$ and $\delta_j = d_{\omega_j^-}^{\Lambda_j^-}(A_j, W)$.

To facilitate the analysis of the paths shown in the figure, let Ω_j be the *family* of situations where each $\hat{\omega}_j \in \Omega_j$ is the same as ω_j^-, except that it also assigns a value to $C_j - A_j$. With this construction, all of the lengths in the figure represented by Greek letters are constant across the situations in Ω_j. Therefore, using the analysis in the proof of Lemma 5, there necessarily exists a single wait edge $(V_j, C_j{:}{-}v_j, A_j)$ such that:

(1) for each $C_j - A_j \in (x_j, y_j]$, at least one of the paths, UV_jA_jW or $UV_jA_jC_jW$, shown in Fig. 38b, has length at most $d_*^{\Lambda_j}(U, W)$;
(2) the maximum length SVP from U to W over situations in Ω_j is obtained when $C_j - A_j = \delta_j - \gamma_j \in (x_j, y_j]$, where $|UV_jA_jW|_{\hat{\omega}_j} = |UV_jA_jC_jW|_{\hat{\omega}_j}$;
(3) that maximum length, call it L, is given by: $\sigma_j + \max\{\delta_j - v_j, \gamma_j\}$;
(4) there exists an SVP from U to V_j comprising solely negative edges; and
(5) there exists an SVP from C_j to W comprising solely non-negative edges.

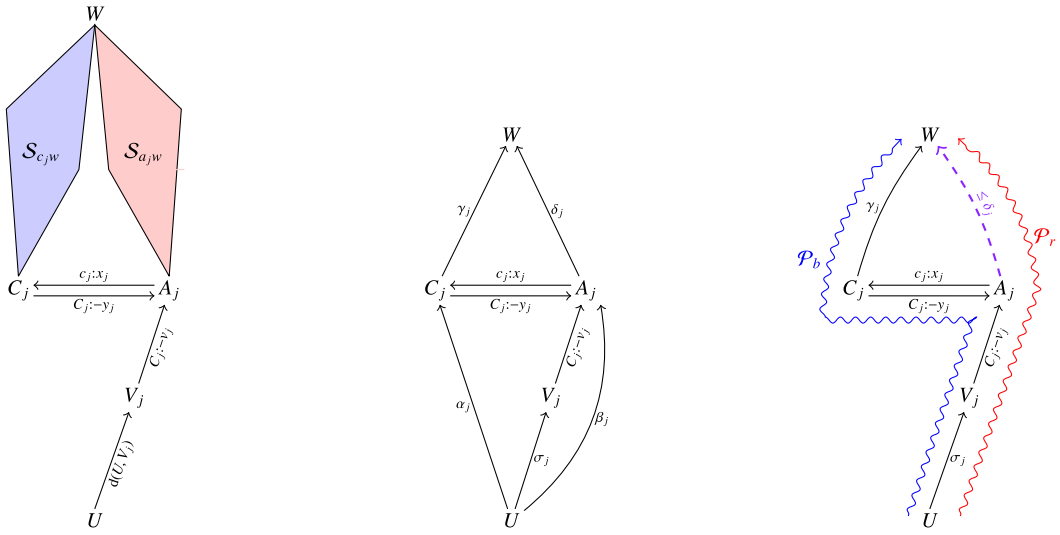
Regarding item (3), by the definition of $d_*^{\Lambda_j}(U, W)$, it follows that $L \le d_*^{\Lambda_j}(U, W)$. Subsequently, it will be shown that $L = d_*^{\Lambda_j}(U, W)$.

Regarding item (4), the choice of CL_j ensures that no activation timepoints from any other contingent link represented in Λ_j^- can appear in the SVP from U to V_j; hence, that path cannot contain any labeled edges and, therefore, must comprise solely ordinary negative edges. Therefore, $d_*^{\Lambda_j^-}(U, V_j) = \mathrm{d}(U, V_j)$.

Regarding item (5), recall that the final subpath along the spine of a canonical structure (e.g., the subpath from A_h to W in $\mathcal{S}_{a_j w}$, seen previously in Fig. 35d) comprises only ordinary edges. Now, if the SVP from C_j to W contained any LC edges, their activation timepoints could not appear in the spine of the structure $\mathcal{S}_{c_j w}$ since the subpath of that spine from U to any activation timepoint comprises only negative edges, whereas the SVP from A_j to C_j to W comprises only non-negative edges. Therefore, any LC edges appearing in the path from C_j to W cannot appear in $\mathcal{S}_{a_j w}$. Similarly, any such LC edges cannot appear in the canonical structures, $\mathcal{S}_{c_j w}$ or $\mathcal{S}_{uw}$. As a result, the durations of any such LC edges cannot affect the paths in any of those canonical structures. Therefore, no loss of generality results from assuming that the situation ω_j^- assigns maximum durations to any such LC edges. But then any LC edges in the path from C_j to W could be replaced by their ordinary stand-in edges, implying that there must be an SVP from C_j to W comprising solely non-negative ordinary edges.

Next, we stipulate that the canonical structure $\mathcal{S}_{uw}^{\Lambda_j}$ shall contain only the one wait edge $(V_j, C_j{:}{-}v_j, A_j)$, given above and as illustrated in Fig. 38a; and we shall show that this structure has all of the needed properties. In particular, we show that in *any* situation ω_j for $\mathcal{G}^{\Lambda_j}$, there exists a simple path $\mathcal{P}$ from U to W whose only edges are drawn from the structure $\mathcal{S}_{uw}^{\Lambda_j}$ and whose projection satisfies $|\mathcal{P}|_{\omega_j} \le L$. Toward that end, let ω_j be any situation for $\mathcal{G}^{\Lambda_j}$; let ω_c be the value of $C_j - A_j$ specified by ω_j; and let $\mathcal{P}_b$ and $\mathcal{P}_r$ be the paths from U to W shown in Fig. 38c, where the paths from U to V_j, and from C_j to W, are shortest paths comprising only ordinary edges. First, for any ω_j for which $\omega_c \le \delta_j - \gamma_j$:

$$
\begin{aligned}
|\mathcal{P}_b|_{\omega_j} &= \sigma_j + \max\{-v_j, -\omega_c\} + \omega_c + \gamma_j \\
&= \sigma_j + \max\{\omega_c + \gamma_j - v_j, \gamma_j\} \\
&\le \sigma_j + \max\{\delta_j - v_j, \gamma_j\} && (\text{Since} \omega_c \le \delta_j - \gamma_j) \\
&= L && (\text{By item (3) above})
\end{aligned}
$$



(a) Canonical structure $\mathcal{S}_{uw}^{\Lambda_j}$

(b) Paths in the projection $\hat{\omega}_j$

(c) Paths in the situation ω_j

Fig. 38. Structures and paths analyzed in the Proof of Claim 3.6.

Second, by Claim 3.4, there must exist a path from A_j to W that uses edges drawn solely from $\mathcal{S}_{a_j w}$ whose projection has length at most $\delta_j = d_*^{\Lambda_j^-}(A_j, W)$, shown as dashed and purple in Fig. 38c. Therefore, for any ω_j for which $\omega_c \geq \delta_j - \gamma_j$:

$$\begin{aligned} |\mathcal{P}_r|_{\omega_j} &\leq \sigma_j + \max\{-v_j, -\omega_c\} + \delta_j \\ &= \sigma_j + \max\{\delta_j - v_j, \delta_j - \omega_c\} \\ &\leq \sigma_j + \max\{\delta_j - v_j, \gamma\} \qquad (\text{Since } \omega_c \geq \delta_j - \gamma_j) \\ &= L \end{aligned}$$

As a result, it follows that $d_*^{\Lambda_j}(U, W) \leq L$. However, since we previously showed that $L \leq d_*^{\Lambda_j}(U, W)$, we now have that $L = d_*^{\Lambda_j}(U, W) = d_*^{\Lambda_j^+}(U, W)$. □

(End of Proof of Theorem 3) □

A.3. Proof of correctness for the `minDisp`$_{\text{ESTNU}}$ algorithm

This section presents a rigorous proof of correctness for the `minDisp`$_{\text{ESTNU}}$ algorithm from Section 7.2.

Corollary 2. *When applied to a dispatchable ESTNU graph $\mathcal{G}$, Algorithm 24 (`genStandIns`) correctly computes $d_*(U, W)$ for each pair of timepoints U and W (i.e., at the end of the algorithm, $d = d_*$).*

Proof. By induction. For any $j \geq 0$, let $Q(j)$ be the following proposition: For any U and W, if Λ_j^+ is any set of labeled edges for which there exists a set Λ_j such that: (1) $\Lambda_j \subseteq \Lambda_j^+$; (2) Λ_j represents j contingent links; and (3) Λ_j is minimally sufficient for $d_*^{\Lambda_j^+}(U, W)$, then `genStandIns` correctly computes $d_*^{\Lambda_j^+}(U, W)$ by its j^{th} iteration.

Let U and W be any timepoints in $\mathcal{G}$; and let Λ_j^+ be any set of labeled edges from $\mathcal{G}$ for which there exists a set Λ_j satisfying properties (1), (2) and (3) above. Choose any such Λ_j.

Base Case: $j = 0$. Then $\Lambda_j = \emptyset$ and $d_*^{\Lambda_j^+}(U, W) = d_*^{\emptyset}(U, W) = \mathrm{d}(U, W)$, which is obtained by the initial call to Johnson's algorithm.

Recursive Case: $j > 0$. Suppose that $Q(h)$ holds for all $h < j$. Since $j > 0$, Theorem 3 provides that there is a contingent link $\mathrm{CL}_j = (A_j, x_j, y_j, C_j)$ and a wait edge $(V_j, C_j{:}{-}v_j, A_j)$ such that $d_*^{\Lambda_j^+}(U, W) = \mathrm{d}(U, V_j) + \max\{\delta_j - v_j, \gamma_j\}$, where: $\delta_j = d_*^{\Lambda_j^-}(A_j, W)$ and $\gamma_j = d_*^{\Lambda_j^-}(C_j, W) = \mathrm{d}(C_j, W)$, where $\Lambda_j^- = \Lambda_j - \mathrm{CL}_j$ is the same as Λ_j, except that it does not include

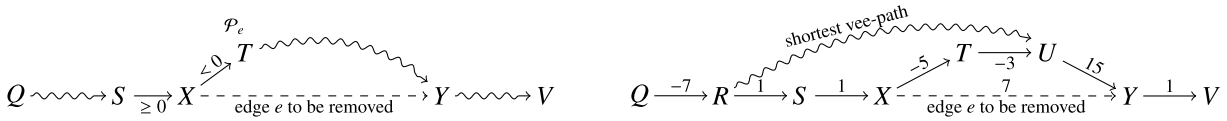


Fig. 39. Scenarios (general, left; specific, right) considered in Case 1 of Lemma 6.

labeled edges associated with CL_j. Since any subset $\Lambda^-_{j-1} \subseteq \Lambda^-_j$ that is minimally sufficient for $d_*^{\Lambda^-_j}(A_j, W)$ necessarily represents fewer than j contingent links, the inductive hypothesis ensures that δ_j has been correctly computed by the $j-1^{\text{st}}$ iteration. In addition, $\gamma_j = \mathrm{d}(C_j, W)$ is given by the initial call to Johnson's algorithm. Therefore, by the j^{th} iteration, $d_*^{\Lambda^+_j}(U, W)$ has been correctly computed.

Therefore, $Q(j)$ holds for all $j \geq 0$. But then for any U and W, $d_*(U, W) = d_*^{\Lambda}(U, W)$ is correctly computed by at most the k^{th} iteration, where Λ contains all of the labeled edges from $\mathcal{G}$, which represent k contingent links. □

Corollary 3. *Given $U, W, \Lambda_j, \Lambda^+_j$, and $\mathcal{S}^{\Lambda_j}_{uw}$ as in the proof of Theorem 3, in* any *situation ω_j for $\mathcal{G}^{\Lambda_j}$, there exists a simple path $\mathcal{P}$ from U to W whose only edges are drawn from the structure $\mathcal{S}^{\Lambda_j}_{uw}$ and whose projection satisfies $|\mathcal{P}|_{\omega_j} \leq d_*^{\Lambda_j}(U, W) = d_*^{\Lambda^+_j}(U, W)$.*

Theorem 4, below, ensures that the final step of `minDisp`$_{\text{ESTNU}}$ (Algorithm 23), namely, removing the wait edges by `markWaits` (Algorithm 27), cannot threaten the dispatchability of the ESTNU. But first, we present an important result about *STN* dispatchability.

Lemma 6. *Suppose that $e = (X, \delta, Y)$ is an (ordinary) edge in a dispatchable* STN *$\mathcal{G}_o$, and $\mathcal{P}_e$ is a shortest vee-path from X to Y in $\mathcal{G}_o$ that does* not *use e. Then removing e from $\mathcal{G}_o$ preserves the dispatchability of $\mathcal{G}_o$.*

Proof. Let $\mathcal{P}_{qv}$ be any SVP from some Q to some V in $\mathcal{G}_o$ that includes e. Since e is an edge in an SVP, it follows that e, by itself, constitutes a shortest path from X to Y. Therefore, the alternative SVP $\mathcal{P}_e$ necessarily satisfies $|\mathcal{P}_e| = |e| = \delta$. Next, let $\mathcal{P}'_{qv}$ be the path obtained from $\mathcal{P}_{qv}$ by replacing e by $\mathcal{P}_e$. By this construction, $|\mathcal{P}'_{qv}| = |\mathcal{P}_{qv}|$; hence, $\mathcal{P}'_{qv}$ is a shortest path. However, suppose that $\mathcal{P}'_{qv}$ is not a vee-path.

Case 1: Replacing e by $\mathcal{P}_e$ inserts a negative edge after a non-negative edge. This scenario is illustrated in general on the lefthand side of Fig. 39, where $\mathcal{P}_e$ is the path from X to T to Y. The righthand side shows a specific example, where $e = (X, 7, Y)$, $\mathcal{P}_{qv}$ is the path *QRSXYV*, $\mathcal{P}_e$ is the path *XTUY*, and replacing e by $\mathcal{P}_e$ inserts the negative edge *XT* after the non-negative edge *SX*. Note that in this (general or specific) case, $\delta = |e|$ must be non-negative since e follows the non-negative edge *SX* in a vee-path. As shown in the specific example, let U be the terminus of the last negative edge in $\mathcal{P}_e$; and let R be the terminus of the last non-negative edge in the subpath from Q to X. (It may happen that $Q = R$.) Since the subpath from X to U comprises only negative edges, that subpath must have negative length. Now, since there is a path from R to U in $\mathcal{G}_o$, the dispatchability of $\mathcal{G}_o$ implies that there must be an SVP from R to U, shown as wavy in the figure. Substituting this (wavy) vee-path for the subpath *RSXTU* in $\mathcal{P}'$, provides an SVP from Q to V that does not use e. (If e were needed by any shortest path $\mathcal{P}_{ru}$ from R to U, then the suffix of $\mathcal{P}_{ru}$ from X to U must be negative, given that *XTU* is a shortest path with $|XTU| < 0$. But then $\mathcal{P}_{ru}$ cannot be a vee-path, since it would include the non-negative edge e followed by one or more negative edges.)

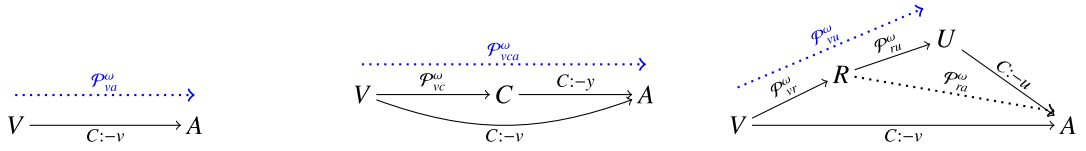
Case 2: Replacing e by $\mathcal{P}_e$ inserts a non-negative edge before a negative edge. Handled similarly. □

Theorem 4. *Algorithm 23 (`minDisp`$_{\text{ESTNU}}$) preserves the dispatchability of its input ESTNU $\mathcal{G}$.*

Proof. (1) The first step of `minDisp`$_{\text{ESTNU}}$ inserts the ordinary edges collected by `genStandIns` (Algorithm 24). Since they are entailed by the network, the addition of these edges cannot disturb dispatchability.

(2) The second step of `minDisp`$_{\text{ESTNU}}$ runs the `disp`$_{\text{STN}}$ algorithm on the ordinary edges, which can insert some ordinary edges while removing others. With no loss of generality, suppose it first inserts all the new edges and then removes edges. Since the edges being inserted are entailed by ordinary paths in the network, their insertion cannot disturb dispatchability. Therefore, let $e = (X, \delta, Y)$ be the first (ordinary) edge whose removal thwarts the ESTNU's dispatchability. Since `disp`$_{\text{STN}}$ preserves dispatchability among the ordinary edges, there must be an alternative (ordinary) SVP $\mathcal{P}_e$ from X to Y with $|\mathcal{P}_e| \leq |e|$. Furthermore, suppose ω is any situation and $\mathcal{P}$ is an ESTNU path that uses e and whose projection in $\mathcal{G}_\omega$ is an SVP. Now, the dispatchability of the ESTNU $\mathcal{G}$ implies that its projection $\mathcal{G}_\omega$ is a dispatchable STN. Therefore, by Lemma 6, replacing e by $\mathcal{P}_e$ in that projection preserves its dispatchability. Since this holds for every projection $\mathcal{G}_\omega$, replacing e by $\mathcal{P}_e$ in $\mathcal{G}$ cannot threaten the dispatchability of $\mathcal{G}$.

(3) The third step of `minDisp`$_{\text{ESTNU}}$ is to remove any remaining stand-in edges.



(a) Case 4.1 (b) Case 4.2 (c) Case 4.3

Fig. 40. Scenarios addressed in item 4 of Theorem 4.

Case 3.1: $\hat{e}$ is a stand-in edge that derives from some individual labeled edge e. By Lemma 3, in any situation ω, $|e|_\omega \leq |\hat{e}|_\omega$. Furthermore, $|\hat{e}|_\omega$ has the same sign as $|e|_\omega$. Therefore, replacing $\hat{e}$ by e in any ESTNU path whose projection is a vee-path will result in an ESTNU path whose projection is a vee-path that can only be shorter or have the same length.
Case 3.2: $\hat{e} = (U, \delta, W)$ is a stand-in edge that derives from some possibly nested diamond structure $\mathcal{S}_{uw}$. In this case, $\delta = d_*(U, W)$. Now, in any situation where the length of an SVP from U to W is less than $|\hat{e}| = \delta = d_*(U, W)$, $\hat{e}$ cannot participate in that SVP and hence is irrelevant. On the other hand, suppose that ω is a situation that is maximal for $d_*(U, W)$ (i.e., where the length of an SVP from U to W equals $\delta = d_*(U, W)$). Now, by Corollary 3, there necessarily exists a simple path $\mathcal{P}$ from U to W whose only edges are drawn from the structure $\mathcal{S}_{uw}$, which does not include $\hat{e}$, and whose projection satisfies $|\mathcal{P}|_\omega \leq d_*(U, W)$. But since ω is maximal for $d_*(U, W)$, that projection cannot be shorter than $d_*(U, W)$; hence, $|\mathcal{P}|_\omega = d_*(U, W) = \delta = |\hat{e}|$. As a result, in *every* projection, there is an SVP from U to W that does not use the edge $\hat{e}$. Then, by Lemma 6, the dispatchability of any STN projection cannot be disturbed by replacing $\hat{e}$ by the corresponding SVP from U to W. Therefore, removing $\hat{e}$ from the ESTNU cannot threaten its dispatchability.

(4) The fourth step of `minDisp`$_{\text{ESTNU}}$ removes several categories of waits collected by `markWaits` (Algorithm 27, Lines 3-6). We consider each category of wait removals in turn. In each case, the wait edge to be removed is denoted by $(V, C{:}{-}v, A)$ and the corresponding contingent link by (A, x, y, C) Recall, too, that once `genStandIns` has been completed, $d = d_*$, which remains fixed for the rest of `minDisp`$_{\text{ESTNU}}$.
Case 4.1: Remove all wait edges $(V, C{:}{-}v, A)$ for which $\mathrm{d}(V, A) \leq -v$. Since `getInitStandins` removed all weak waits (Line 5), it follows that $-v < -x$, where $-x$ is the length of the stand-in edge for $(V, C{:}{-}v, A)$. Now, after calling `disp`$_{\text{STN}}$, which does not affect $d = d_*$, but before removing any remaining stand-in edges, there must have been a simple *ordinary* SVP $\mathcal{P}_{va}$ from V to A of length $\delta = \mathrm{d}(V, A) \leq -v$, some of whose edges may have been stand-in edges, whether associated with individual labeled edges or possibly-nested diamond structures. Furthermore, since $-v < 0$, that SVP must have started with at least one negative edge; and that first negative edge could not be the stand-in edge $(V, -x, A)$ for $(V, C{:}{-}v, A)$, since the suffix from A to A would have to comprise only zero-length edges and, in any case, the total length of the ordinary path would be $-x > -v$. Now, by Cases 3.1 and 3.2 above, in every situation ω, there exist replacement edges or paths for each of the stand-in edges in $\mathcal{P}_{va}$ that together ensure the existence of an alternative ESTNU path $\mathcal{P}^{\omega}_{va}$ from V to A, shown as blue and dotted in Fig. 40a, whose projection in the situation ω is an SVP. Furthermore, none of the replacements done in Cases 3.1 or 3.2 affect the first negative edge in the original ordinary SVP from V to A. Therefore, $\mathcal{P}^{\omega}_{va}$ cannot include the wait edge $(V, C{:}{-}v, A)$, since it would create a negative-length path from V to V.
Case 4.2: Remove all wait edges $(V, C{:}{-}v, A)$ for which $\mathrm{d}(V, C) < 0$. Let $(V, C{:}{-}v, A)$ be the first such wait edge whose removal thwarts the dispatchability of the ESTNU. After calling `disp`$_{\text{STN}}$, which does not affect $d = d_*$, but before the third step's removal of stand-in edges, there must have been an ordinary SVP $\mathcal{P}_{vc}$ from V to C, some of whose edges may be stand-in edges, whether arising from individual labeled edges or possibly-nested diamond structures. However, by Cases 3.1 and 3.2 above, in every situation ω, there exist replacement edges or paths for each of those stand-in edges that together ensure the existence of an alternative ESTNU path $\mathcal{P}^{\omega}_{vc}$ from V to C, illustrated in Fig. 40b, that does not use any stand-in edges and whose projection in the situation ω is an SVP. Next, let $\mathcal{P}^{\omega}_{vca}$ be the ESTNU path, shown as blue and dotted in the figure, obtained by concatenating $\mathcal{P}^{\omega}_{vc}$ and the UC edge $(C, C{:}{-}y, A)$. Then $|\mathcal{P}^{\omega}_{vca}|_\omega = \mathrm{d}(V, C) - \omega_c < -\omega_c \leq \max\{-v, -\omega_c\} = |(V, C{:}{-}v, A)|_\omega$. (As seen elsewhere, ω_c denotes the value of $C - A$ in the situation ω.) Therefore, the projection of $\mathcal{P}^{\omega}_{vca}$ is a *shorter* path than the projection of $(V, C{:}{-}v, A)$. Although $\mathcal{P}^{\omega}_{vca}$ might not be an SVP, the dispatchability of $\mathcal{G}_\omega$ ensures that there must be an SVP from V to A whose length is at most $|\mathcal{P}^{\omega}_{vca}|_\omega < |(V, C{:}{-}v, A)|_\omega$. In turn, that implies that $(V, C{:}{-}v, A)$ cannot participate in that SVP. Therefore, removing $(V, C{:}{-}v, A)$ from the ESTNU cannot affect the dispatchability of $\mathcal{G}_\omega$. And since the choice of ω was arbitrary, removing that edge cannot affect the ESTNU's dispatchability.
Case 4.3: Remove each wait edge $(V, C{:}{-}v, A)$ for which there is a wait edge $(U, C{:}{-}u, A)$ with $\mathrm{d}(V, U) < 0$ and $\mathrm{d}(V, U) - u \leq -v$. Let $(V, C{:}{-}v, A)$ be the first such wait edge whose removal thwarts the dispatchability of the ESTNU. As in Cases 4.1 and 4.2, after running `disp`$_{\text{STN}}$, but before removing any remaining stand-in edges, there must have been an ordinary SVP $\mathcal{P}_{vu}$ from V to U, some of whose edges may have been stand-in edges. However, for each situation ω, Cases 3.1 and 3.2 above ensure the existence of replacement edges or paths for those stand-in edges that together ensure the existence of an alternative ESTNU path $\mathcal{P}^{\omega}_{vu}$ from V to U, shown as blue and dotted in Fig. 40c, that does not use any stand-in edges and whose projection in the situation ω is an SVP. Hence, $|\mathcal{P}^{\omega}_{vu}|_\omega \leq \mathrm{d}(V, U) < 0$. Since

$|\mathcal{P}^{\omega}_{vu}|_{\omega} < 0$, that SVP must start with at least one negative edge. Next, let R be the terminus of the last negative edge in $\mathcal{P}^{\omega}_{vu}$; let $\mathcal{P}^{\omega}_{vr}$ be the prefix of $\mathcal{P}^{\omega}_{vu}$ from V to R; and let $\mathcal{P}^{\omega}_{ru}$ be the suffix of $\mathcal{P}^{\omega}_{vu}$ from R to U. Now, since there is a path from R to A, the dispatchability of $\mathcal{G}_{\omega}$ ensures that there is an SVP from R to A. Let $\mathcal{P}^{\omega}_{ra}$ be an ESTNU path whose projection in ω is that SVP from R to A, shown as dotted in the figure. Next, let $\mathcal{P}^{\omega}_{vra}$ be the concatenation of $\mathcal{P}^{\omega}_{vr}$ and $\mathcal{P}^{\omega}_{ra}$. Then the projection of $\mathcal{P}^{\omega}_{vra}$ is a vee-path for which

$$
\begin{array}{rcll}
|\mathcal{P}^{\omega}_{vra}|_{\omega} & = & |\mathcal{P}^{\omega}_{vr}|_{\omega} + |\mathcal{P}^{\omega}_{ra}|_{\omega} & \\
& \leq & |\mathcal{P}^{\omega}_{vr}|_{\omega} + (|\mathcal{P}^{\omega}_{ru}|_{\omega} + |(U, C{:}{-}u, A)|_{\omega_c}) & \text{(Since } \mathcal{P}^{\omega}_{ra} \text{ is a shortest path)} \\
& = & |\mathcal{P}^{\omega}_{vu}|_{\omega} + |(U, C{:}{-}u, A)|_{\omega_c} & \text{(Since } \mathcal{P}^{\omega}_{vu} \text{ is the concatenation of } \mathcal{P}^{\omega}_{vr} \text{ and } \mathcal{P}^{\omega}_{ru}) \\
& \leq & \mathrm{d}(V, U) + \max\{-u, -\omega_c\} & \\
& \leq & \max\{\mathrm{d}(V, U) - u, \mathrm{d}(V, U) - \omega_c\} & \\
& \leq & \max\{-v, -\omega_c\} & \text{(Since } \mathrm{d}(V, U) - u \leq -v \text{ and } \mathrm{d}(V, U) < 0) \\
& = & |(V, C{:}{-}v, A)|_{\omega_c}. &
\end{array}
$$

Furthermore, the SVP from R to A cannot include $(V, C{:}{-}v, A)$ since that would imply a chain of negative edges from V to V within the vee-path $\mathcal{P}^{\omega}_{vra}$, contradicting the dispatchability of the network. Therefore, $\mathcal{P}^{\omega}_{vra}$ is an alternative vee-path that is at least as short as the wait edge $(V, C{:}{-}v, A)$. Therefore, the removal of $(V, C{:}{-}v, A)$ cannot thwart the dispatchability of the ESTNU. □

Theorem 5. *When given a dispatchable ESTNU as input, the* $\texttt{minDisp}_{\text{ESTNU}}$ *algorithm outputs an* equivalent *dispatchable ESTNU having a* minimal *number of edges (i.e., a μESTNU).*

Proof. The ESTNU output by $\texttt{minDisp}_{\text{ESTNU}}$ is equivalent to the input network since: (1) all fixed waits and stand-in edges are entailed by alternative edges or paths, (2) the $\texttt{disp}_{\text{STN}}$ algorithm generates an equivalent set of ordinary edges, and (3) each removed wait edge is entailed by an alternative path in every situation.

Minimality. Since Theorem 4 ensures that the output ESTNU is dispatchable, it suffices to show that each edge in the output ESTNU is needed for dispatchability (i.e., its removal would thwart dispatchability). To the contrary, suppose that e is some edge *not* needed for dispatchability but belongs to the output ESTNU and, hence, was not removed. For reference, let $\mathcal{G}_o = (\mathcal{T}, \mathcal{E}^*_o)$ be the STN output by $\texttt{disp}_{\text{STN}}$.

Case 1: e is an ordinary edge (U, δ, W). Since all stand-in edges are eventually removed, e cannot be a stand-in edge. Instead, it must be an original ordinary edge or one inserted by $\texttt{disp}_{\text{STN}}$, with $\delta = |e| = d_*(U, W)$. Suppose some set Λ of labeled edges was minimally sufficient for $d_*(U, W)$. If the number of contingent links represented in Λ were greater than zero, then there would be a canonical structure of nested diamonds that would provide $d_*(U, W)$, contradicting that e was not a stand-in edge. Therefore, $\Lambda = \emptyset$ and $d_*(U, W) = \mathrm{d}(U, W) = \delta$. (Recall that $d_* = d$ at this point in the algorithm.) But $\texttt{disp}_{\text{STN}}$ outputs a minimal dispatchable STN, keeping or inserting e, from which it follows that any vee-path $\mathcal{P}$ from U to W in $\mathcal{G}_o$ that does not use e must satisfy $|\mathcal{P}| > \delta$. But that implies that removing e from the ESTNU would yield $d_*(U, W) > \delta = |e|$, contradicting that e is not needed.

Case 2: e is a wait edge $(V, C{:}{-}v, A)$ *associated with a contingent link* $CL = (A, x, y, C)$. Since e was not removed, the conditions in Lines 3-6 of $\texttt{markWaits}$ must all be false; hence:

(f1) $d_*(V, A) > -v$;
(f2) $d_*(V, C) \geq 0$; and
(f3) for all *other* waits $(U, C{:}{-}u, A)$, $d_*(V, U) \geq 0$ or $d_*(V, U) - u > -v$.

Let Λ^- denote the set of all labeled edges from $\mathcal{G}$ except those associated with CL; and let $\mathcal{S}^{\Lambda^-}_{va}$ be the canonical structure associated with $d^{\Lambda^-}_*(V, A)$ that is guaranteed by Theorem 3. Let ω be the situation for which (1) $\omega_c = C - A = y$ (i.e., its maximum value); (2) the duration of each contingent link $CL_i = (A_i, x_i, y_i, C_i)$ having labeled edges in the canonical structure $\mathcal{S}^{\Lambda^-}_{va}$ is the value $\delta_i - \gamma_i$ specified in the proof of Theorem 3, where all such durations combine to ensure that ω is maximal for $d^{\Lambda^-}_*(V, A)$; and (3) the duration of every other contingent link (A_g, x_g, y_g, C_g) is its minimum value, x_g.

Since e is not needed to preserve the dispatchability of the ESTNU $\mathcal{G}$, then in the situation ω, there must be a simple ESTNU path $\mathcal{P}^{va}$ from V to A that does not use e, and whose projection satisfies $|\mathcal{P}^{va}|_{\omega} \leq |e|_{\omega} = \max\{-\omega_c, -v\} = \max\{-y, -v\} = -v$. Being simple, $\mathcal{P}^{va}$ cannot include the LC edge $(A, c{:}x, C)$ associated with CL; and if it includes the UC edge $(C, C{:}{-}y, A)$ or some wait $(U, C{:}{-}u, A)$ associated with CL, then that edge must be the last edge in $\mathcal{P}^{va}$.

Case 2a: The last edge in $\mathcal{P}^{va}$ *is the UC edge* $(C, C{:}{-}y, A)$. Since the projection of a negative edge is necessarily negative, then the last edge in the projection of $\mathcal{P}^{va}$ is negative. And since its projection is an SVP, it follows that *every* edge in the projection is negative and, hence, so is every edge in $\mathcal{P}^{va}$. Let $\mathcal{P}^{vc}$ be the subpath of $\mathcal{P}^{va}$ from V to C. Since all of its edges are negative, then in every situation the projection of $\mathcal{P}^{vc}$ must have negative length. Therefore, in every situation, every SVP from V to C must have negative length, implying that $d_*(V, C) < 0$, contradicting condition (f2), above.

Case 2b: The last edge in $\mathcal{P}^{va}$ *is a wait edge* $(U, C{:}{-}u, A)$ *associated with the contingent link CL.* Let $\mathcal{P}^{vu}$ be the prefix of $\mathcal{P}^{va}$ from V to U. As in Case 2a, it follows that every edge in $\mathcal{P}^{vu}$ and its SVP projection must be negative, and hence $d_*(V, U) < 0$. In addition, since $-v \geq |\mathcal{P}^{va}|_{\omega} = |\mathcal{P}^{vu}|_{\omega} - u$, we get that $|\mathcal{P}^{vu}|_{\omega} \leq u - v$.

Next, consider any activation timepoint A_i for some contingent link $CL_i = (A_i, x_i, y_i, C_i)$ that appears in $\mathcal{P}^{va}$. By the same logic, $d_*(A_i, A) < 0$. On the other hand, if CL_i is one of the contingent links having edges in the canonical structure $\mathcal{S}^{\Lambda^-}_{va}$, then by the proof of Theorem 3, $\delta_i = d_*(A_i, A)$; and $\delta_i - \gamma_i \in (x_i, y_i]$ implies that $\delta_i \geq \gamma_i + x_i > 0$. In other words, the contingent links having edges in $\mathcal{S}^{\Lambda^-}_{va}$ cannot have any labeled edges in $\mathcal{P}^{va}$ (or $\mathcal{P}^{vu}$). Hence, the UC or wait edges in $\mathcal{P}^{vu}$ necessarily take on their minimum (i.e., *weakest*) values in the situation ω specified above.

But then in *any* situation $\hat{\omega}$, the projections of those UC or wait edges must have lengths that are shorter than or the same as the lengths of their projections in ω. Hence: $|\mathcal{P}^{vu}|_{\hat{\omega}} \leq |\mathcal{P}^{vu}|_{\omega} \leq u - v$, which implies that every SVP from V to U must have length at most $u - v$, which in turn implies that $d_*(V, U) \leq u - v$ (i.e., $d_*(V, U) - u \leq -v$). Together with $d_*(V, U) < 0$, this contradicts condition (f3), above.

Case 2c: $\mathcal{P}^{va}$ does not contain any labeled edges associated with $CL = (A, x, y, C)$. Since ω is maximal for $d^{\Lambda^-}_*(V, A)$, it follows that in any situation $\hat{\omega}$, $|\mathcal{P}^{va}|_{\hat{\omega}} \leq |\mathcal{P}^{va}|_{\omega} \leq -v$, which implies that any SVP from V to A must have length at most $-v$, which implies that $d_*(V, A) \leq -v$, contradicting condition (f1), above. □

References

[1] M. Ai-Chang, J. Bresina, L. Charest, A. Chase, J.C.J. Hsu, A. Jhonson, B. Kanefsky, P. Morris, K. Rajan, J. Yglesias, B.G. Chafin, W.C. Dias, P.F. Maldague, Mapgeri: mixed-initiative planning and scheduling for the Mars exploration rover mission, IEEE Intell. Syst. 19 (2004) 8–12, https://doi.org/10.1109/MIS.2004.1265878.

[2] S. Akmal, S. Ammons, H. Li, J.C. Boerkoel Jr., Quantifying degrees of controllability in temporal networks with uncertainty, in: 29th International Conference on Automated Planning and Scheduling (ICAPS 2019), 2019, pp. 22–30.

[3] S. Akmal, S. Ammons, H. Li, M. Gao, L. Popowski, J.C. Boerkoel Jr., Quantifying controllability in temporal networks with uncertainty, Artif. Intell. 289 (2020) 103–384, https://doi.org/10.1016/j.artint.2020.103384.

[4] L. Antonyshyn, J. Silveira, S. Givigi, J. Marshall, Multiple mobile robot task and motion planning: a survey, ACM Comput. Surv. 55 (2022), https://doi.org/10.1145/3564696.

[5] N. Bhargava, Multi-agent coordination under uncertain communication, in: 33rd AAAI Conference on Artificial Intelligence (AAAI-19), vol. 33, 2019, pp. 9878–9879.

[6] N. Bhargava, T. Vaquero, B.C. Williams, Faster conflict generation for dynamic controllability, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17), 2017, pp. 4280–4286.

[7] N. Bhargava, B.C. Williams, Complexity bounds for the controllability of temporal networks with conditions, disjunctions, and uncertainty, Artif. Intell. 271 (2019) 1–17, https://doi.org/10.1016/J.ARTINT.2018.11.008.

[8] S.C. Brailsford, C.N. Potts, B.M. Smith, Constraint satisfaction problems: algorithms and applications, Eur. J. Oper. Res. 119 (1999) 557–581, https://doi.org/10.1016/S0377-2217(98)00364-6.

[9] 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://cdn.aaai.org/ICAPS/2005/ICAPS05-005.pdf.

[10] G.S. Brodal, G. Lagogiannis, R.E. Tarjan, Strict Fibonacci heaps, ACM Trans. Algorithms 21 (2025) 1–18, https://doi.org/10.1145/3707692.

[11] V. Brusoni, L. Console, P. Terenziani, B. Pernid, Later: managing temporal information efficiently, IEEE Expert 12 (1997) 56–64, https://doi.org/10.1109/64.608197.

[12] M. Cairo, L. Hunsberger, R. Rizzi, Faster dynamic controllablity checking for simple temporal networks with uncertainty, in: 25th International Symposium on Temporal Representation and Reasoning (TIME 2018), 2018, pp. 8:1–8:16.

[13] M. Cairo, R. Rizzi, Dynamic controllability made simple, in: 24th International Symposium on Temporal Representation and Reasoning (TIME 2017), 2017, pp. 8:1–8:16.

[14] A. Cesta, G. Cortellessa, R. Rasconi, F. Pecora, M. Scopelliti, L. Tiberio, Monitoring elderly people with the ROBOCARE domestic environment: interaction synthesis and user evaluation, in: Comput. Intell., 2011, pp. 60–82.

[15] S. Chien, B. Smith, G. Rabideau, N. Muscettola, K. Rajan, Automated planning and scheduling for goal-based autonomous spacecraft, IEEE Intell. Syst. Appl. 13 (1998) 50–55, https://doi.org/10.1109/5254.722362.

[16] A. Cimatti, A. Micheli, M. Roveri, Solving strong controllability of temporal problems with uncertainty using SMT, Constraints 20 (2014) 1–29, https://doi.org/10.1007/s10601-014-9167-5.

[17] 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.

[18] C. Combi, L. Viganò, M. Zavatteri, Security constraints in temporal role-based access-controlled workflows, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy (CODASPY'16), ACM, 2016, pp. 207–218.

[19] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 4th edition, MIT Press, 2022, https://mitpress.mit.edu/9780262046305/introduction-to-algorithms.

[20] M. De Ryck, D. Pissoort, T. Holvoet, E. Demeester, Decentral task allocation for industrial AGV-systems with routing constraints, J. Manuf. Syst. 62 (2022) 135–144, https://doi.org/10.1016/j.jmsy.2021.11.012.

[21] 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.

[22] 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.

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

[24] 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.

[25] X. Feng, H. Zuo, Q. Sun, Research on collaborative scheduling of aircraft ground service vehicles based on simple temporal network, in: 2021 IEEE 3rd Int. Conf. on Civil Aviation Safety and Information Technology (ICCASIT), 2021, pp. 263–269.

[26] M. Franceschetti, J. Eder, Checking temporal service level agreements for web service compositions with temporal parameters, in: IEEE Int. Conf. on Web Services (ICWS-19), 2019, pp. 443–445.

[27] M. Franceschetti, J. Eder, Negotiating temporal commitments in cross-organizational business processes, in: 27th International Symposium on Temporal Representation and Reasoning (TIME 2020), Dagstuhl Publisher, 2020, pp. 4:1–4:15.

[28] M. Ghallab, D. Nau, P. Traverso, Automated Planning: Theory and Practice, Elsevier, 2004.

[29] K.v.d. Houten, L. Planken, E. Freydell, D.M.J. Tax, M.d. Weerdt, Proactive and reactive constraint programming for stochastic project scheduling with maximal time-lags, in: Proceedings of the 39th AAAI Conference on Artificial Intelligence, 2025, pp. 26534–26541.

[30] 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.
[31] L. Hunsberger, A fast incremental algorithm for managing the execution of dynamically controllable temporal networks, in: N. Markey, J. Wijsen (Eds.), 17th International Symposium on Temporal Representation and Reasoning (TIME 2010), 2010, pp. 121–128.
[32] L. Hunsberger, A faster execution algorithm for dynamically controllable STNUs, in: C. Sanchez, K.B. Venable, E. Zimanyi (Eds.), 20th International Symposium on Temporal Representation and Reasoning (TIME 2013), IEEE CSP, 2013.
[33] L. Hunsberger, Magic loops in simple temporal networks with uncertainty–exploiting structure to speed up dynamic controllability checking, in: J. Filipe, A.L.N. Fred (Eds.), 5th International Conference on Agents and Artificial Intelligence (ICAART 2013), 2013, pp. 157–170.
[34] L. Hunsberger, Magic loops and the dynamic controllability of simple temporal networks with uncertainty, in: J. Filipe, A. Fred (Eds.), Agents and Artificial Intelligence, 2014, pp. 332–350.
[35] L. Hunsberger, Efficient execution of dynamically controllable simple temporal networks with uncertainty, Acta Inform. 53 (2015) 89–147, https://doi.org/10.1007/s00236-015-0227-0.
[36] 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 Pres, 2022, pp. 9776–9785.
[37] L. Hunsberger, R. Posenato, A faster algorithm for converting simple temporal networks with uncertainty into dispatchable form, Inf. Comput. 293 (2023) 1–21, https://doi.org/10.1016/j.ic.2023.105063.
[38] L. Hunsberger, R. Posenato, A faster algorithm for finding negative cycles in simple temporal networks with uncertainty, in: P. Sala, M. Sioutis, F. Wang (Eds.), 31st International Symposium on Temporal Representation and Reasoning (TIME 2024), Dagstuhl Publisher, 2024, pp. 9:1–9:15.
[39] L. Hunsberger, R. Posenato, Converting simple temporal networks with uncertainty into minimal equivalent dispatchable form, in: Proceedings of the Thirty-Fourth International Conference on Automated Planning and Scheduling (ICAPS 2024), 2024, pp. 290–300.
[40] L. Hunsberger, R. Posenato, Faster algorithm for converting an STNU into minimal dispatchable form, in: P. Sala, M. Sioutis, F. Wang (Eds.), 31st International Symposium on Temporal Representation and Reasoning (TIME 2024), Dagstuhl Publisher, 2024, pp. 11:1–11:14.
[41] L. Hunsberger, R. Posenato, Foundations of dispatchability for simple temporal networks with uncertainty, in: 16th International Conference on Agents and Artificial Intelligence (ICAART 2024), SCITEPRESS, 2024, pp. 253–263.
[42] L. Hunsberger, R. Posenato, Robust execution of probabilistic STNs, in: P. Sala, M. Sioutis, F. Wang (Eds.), 31st International Symposium on Temporal Representation and Reasoning (TIME 2024), Dagstuhl Publisher, 2024, pp. 12:1–12:19.
[43] L. Hunsberger, R. Posenato, A better algorithm for converting an STNU into minimal dispatchable form, in: Proceedings of the 32nd International Symposium on Temporal Representation and Reasoning (TIME 2025), Dagstuhl Publisher, 2025, pp. 1–15.
[44] L. Hunsberger, R. Posenato, Canonical Form of Nested Diamond Structures, Technical Report 111/2025, Dipartimento di Informatica - Università degli Studi di Verona, 2025, https://iris.univr.it/handle/11562/1163111.
[45] F. Ingrand, S. Lemai-Chenevier, F. Py, Decisional autonomy of planetary rovers, J. Field Robot. 24 (2007) 559–580, https://doi.org/10.1002/rob.20206.
[46] A.K. Jonsson, P.H. Morris, N. Muscettola, K. Rajan, Planning in interplanetary space: theory and practice, in: Proceeding of AIPS 2000, 2000, p. 10, https://www.aaai.org/Papers/AIPS/2000/AIPS00-019.pdf.
[47] 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.
[48] A. Lanz, R. Posenato, C. Combi, M. Reichert, Controlling time-awareness in modularized processes, in: Enterprise, Business-Process and Information Systems Modeling. BPMDS 2016, EMMSAD 2016, Sprin, 2016, pp. 157–172.
[49] M. Lombardi, M. Milano, L. Benini, Robust scheduling of task graphs under execution time uncertainty, IEEE Trans. Comput. 62 (2013) 98–111, https://doi.org/10.1109/TC.2011.203.
[50] M. Maniadakis, E. Hourdakis, P. Trahanias, Time-informed task planning in multi-agent collaboration, Cogn. Syst. Res. (2016), https://doi.org/10.1016/j.cogsys.2016.09.004.
[51] R. Masson, F. Lehuédé, O. Péton, The Dial-A-Ride problem with transfers, Comput. Oper. Res. 41 (2014) 12–23, https://doi.org/10.1016/j.cor.2013.07.020.
[52] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, R. McEwen, A deliberative architecture for AUV control, in: Proc. - IEEE Int. Conf. Robot. Autom., 2008, pp. 1049–1054.
[53] P. Morris, A structural characterization of temporal dynamic controllability, in: Principles and Practice of Constraint Programming (CP-2006), 2006, pp. 375–389.
[54] P. Morris, Dynamic controllability and dispatchability relationships, in: Int. Conf. on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR-2014), in: LNCS, vol. 8451, Springer, 2014, pp. 464–479.
[55] P. Morris, The mathematics of dispatchability revisited, in: 26th International Conference on Automated Planning and Scheduling (ICAPS-2016), AAAI Press, 2016, pp. 244–252.
[56] P. Morris, N. Muscettola, T. Vidal, Dynamic control of plans with temporal uncertainty, in: 17th Int. Joint Conf. on Artificial Intelligence (IJCAI-2001), 2001, pp. 494–499, https://www.ijcai.org/Proceedings/01/IJCAI-2001-e.pdf.
[57] P.H. Morris, N. Muscettola, Managing temporal uncertainty through waypoint controllability, in: 7th Int. Joint Conf. on Artificial Intelligence (IJCAI-1999), 1999, pp. 1193–1198, https://www.ijcai.org/Proceedings/99-2/Papers/083.pdf.
[58] 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.
[59] 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.
[60] N. Muscettola, P.H. Morris, I. Tsamardinos, Reformulating temporal plans for efficient execution, in: Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kaufmann, 1998, pp. 444–452.
[61] OMG Object Management Group, Business process definition metamodel (bpdm), Beta 1, http://www.omg.org, 2007.
[62] 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.
[63] R. Posenato, STNU Benchmark version 2020, https://profs.scienze.univr.it/~posenato/software/cstnu/benchmarkWrapper, 2020.
[64] R. Posenato, CSTNU tool: a Java library for checking temporal networks, SoftwareX 17 (2022) 100905, https://doi.org/10.1016/j.softx.2021.100905.
[65] R. Posenato, A. Lanz, C. Combi, M. Reichert, Managing time-awareness in modularized processes, Softw. Syst. Model. 18 (2019) 1135–1154, https://doi.org/10.1007/s10270-017-0643-4.
[66] Y. Qi, Y. Liu, D. Gu, J. Zhu, An algorithm of dynamic temporal constraints for the mission series in deep space detectors, Adv. Space Res. 73 (2023) 3855–3867, https://doi.org/10.1016/j.asr.2023.08.037.
[67] C. Strassmair, N.K. Taylor, Human robot collaboration in production environments, in: 23rd IEEE International Symposium on Robot and Human Interactive Communication 2014, 2014, pp. 1–6, https://researchportal.hw.ac.uk/en/publications/human-robot-collaboration-in-production-environments.
[68] A. Sumic, T. Vidal, A more efficient and informed algorithm to check weak controllability of simple temporal networks with uncertainty, in: 31st International Symposium on Temporal Representation and Reasoning (TIME 2024), Dagstuhl Publisher, 2024, pp. 8:1–8:15.
[69] A. Sumic, T. Vidal, A. Micheli, A. Cimatti, Introducing interdependent simple temporal networks with uncertainty for multi-agent temporal planning, in: 31st International Symposium on Temporal Representation and Reasoning (TIME 2024), Dagstuhl Publishing, 2024, pp. 13:1–13:14.

[70] R.E. Tarjan, Shortest Paths, Technical Report, AT&T Bell Laboratories, 1981.
[71] I. Tsamardinos, A probabilistic approach to robust execution of temporal plans with uncertainty, in: Methods and Applications of Artificial Intelligence (SETN 2002), 2002, pp. 97–108.
[72] I. Tsamardinos, N. Muscettola, P. Morris, Fast transformation of temporal plans for efficient execution, in: 15th National Conf. on Artificial Intelligence (AAAI-1998), The MIT Press, 1998, pp. 254–261, https://cdn.aaai.org/AAAI/1998/AAAI98-035.pdf.
[73] T. Vidal, H. Fargier, Handling contingency in temporal constraint networks: from consistency to controllabilities, J. Exp. Theor. Artif. Intell. 11 (1999) 23–45, https://doi.org/10.1080/095281399146607.
[74] A.J. Wang, Risk-bounded Dynamic Scheduling of Temporal Plans, Ph.D. thesis, Massachusetts Institute of Technology, 2022, https://hdl.handle.net/1721.1/147542.
[75] H.L.S. Younes, R.G. Simmons, VHPOP: versatile heuristic partial order planner, J. Artif. Intell. Res. 20 (2003) 405–430, https://doi.org/10.1613/jair.1136.
[76] P. Yu, C. Fang, B.C. Williams, Resolving uncontrollable conditional temporal problems using continuous relaxations, in: 24th International Conference on Automated Planning and Scheduling, ICAPS 2014, AAAI, 2014, pp. 341–349.
[77] P. Yu, B.C. Williams, C. Fang, J. Cui, P. Haslum, Resolving over-constrained temporal problems with uncertainty through conflict-directed relaxation, J. Artif. Intell. Res. 60 (2017) 425–490, https://doi.org/10.1613/jair.5431.