# Faster Algorithm for Converting an STNU into Minimal Dispatchable Form

**Luke Hunsberger**
Vassar College, Poughkeepsie, NY, USA

**Roberto Posenato**
University of Verona, Italy

**Abstract**

A Simple Temporal Network with Uncertainty (STNU) is a data structure for representing and reasoning about temporal constraints on activities, including those with uncertain durations. An STNU is dispatchable if it can be flexibly and efficiently executed in real time while guaranteeing that all relevant constraints are satisfied. The number of edges in a dispatchable network affects the computational work that must be done during real-time execution. Recent work presented an $O(kn^3)$-time algorithm for converting a dispatchable STNU into an equivalent dispatchable network having a minimal number of edges, where $n$ is the number of timepoints and $k$ is the number of actions with uncertain durations. This paper presents a modification of that algorithm, making it an order of magnitude faster, down to $O(n^3)$. Given that in typical applications $k = O(n)$, this represents an effective order-of-magnitude reduction from $O(n^4)$ to $O(n^3)$.

**2012 ACM Subject Classification** Computing methodologies → Temporal reasoning; Theory of computation → Dynamic graph algorithms

**Keywords and phrases** Temporal constraint networks, dispatchable networks

**Digital Object Identifier** 10.4230/LIPIcs.TIME.2024.11

## 1 Background

Temporal constraint networks facilitate representing and reasoning about temporal constraints on activities. *Simple Temporal Networks with Uncertainty* (STNUs) are one of the most important kinds of temporal networks because they allow the explicit representation of actions with uncertain durations [13]. An STNU is *dispatchable* if it can be executed by a flexible and efficient real-time execution algorithm while guaranteeing that all of its constraints will be satisfied. This paper modifies an existing algorithm for converting a dispatchable network into an equivalent dispatchable network having a minimal number of edges, making it an order of magnitude faster.

## 1.1 Simple Temporal Networks

A *Simple Temporal Network* (STN) is a pair $(\mathcal{T}, \mathcal{C})$ where $\mathcal{T}$ is a set of real-valued variables called timepoints; and $\mathcal{C}$ is a set of *ordinary* constraints, each of the form $(Y - X \leq \delta)$ for $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$ [3]. An STN is *consistent* if it has a solution as a constraint satisfaction problem (CSP). Each STN has a corresponding graph where the timepoints serve as nodes and the constraints correspond to labeled, directed edges. In particular, each constraint $(Y - X \leq \delta)$ corresponds to an edge $X \xrightarrow{\delta} Y$ in the graph. For convenience, such edges may be notated as $(X, \delta, Y)$ or, if the weight is not being considered, simply $XY$. Similarly, a path from $X$ to $Y$ may be notated by listing the timepoints visited by the path (e.g., $XUVWY$) or, if the context is clear, simply $XY$.

A flexible and efficient *real-time execution* (RTE) algorithm has been defined for STNs that maintains time windows for each timepoint and, as each timepoint $X$ is executed, only propagates constraints *locally,* to *neighbors* of $X$ in the STN graph [16, 14]. An STN is called *dispatchable* if that RTE algorithm is guaranteed to satisfy all of the STN's constraints no matter how the flexibility afforded by the algorithm is exploited during execution. Morris [12] proved that a consistent STN is dispatchable if and only if every pair of timepoints that are connected by a path in the STN graph are connected by a shortest *vee-path* (i.e., a shortest path comprising zero or more negative edges followed by zero or more non-negative edges). Algorithms for generating equivalent dispatchable STNs having a *minimal number of edges* have been presented [16, 14]. Minimizing the number of edges is important since it directly impacts the real-time computations required during execution.

## 1.2 Simple Temporal Networks with Uncertainty

A *Simple Temporal Network with Uncertainty* (STNU) augments an STN to include *contingent links* that represent actions with uncertain, but bounded durations [13]. 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 of the form $(A, x, y, C)$, where $A, C \in \mathcal{T}$ and $0 < x < y < \infty$. The semantics of STNU execution ensures that regardless of when the *activation timepoint* $A$ is executed, the *contingent timepoint* $C$ will occur such that $C - A \in [x, y]$. Thus, the duration $C - A$ is uncontrollable, but bounded. Each STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ has a corresponding graph $\mathcal{G} = (\mathcal{T}, \mathcal{E}_{\text{o}}, \mathcal{E}_{\text{lc}}, \mathcal{E}_{\text{uc}})$, where $(\mathcal{T}, \mathcal{E}_{\text{o}})$ is the graph for the STN $(\mathcal{T}, \mathcal{C})$, and $\mathcal{E}_{\text{lc}}$ and $\mathcal{E}_{\text{uc}}$ are sets of *labeled* edges corresponding to the contingent durations in $\mathcal{L}$. In particular, each contingent link $(A, x, y, C)$ in $\mathcal{L}$ has a *lower-case* (LC) edge $A \xrightarrow{c:x} C$ in $\mathcal{E}_{\text{lc}}$ that represents the uncontrollable possibility that the duration might take on its minimum value $x$; and an *upper-case* (UC) edge $C \xrightarrow{C:-y} A$ in $\mathcal{E}_{\text{uc}}$ that represents the possibility that it might take on its maximum value $y$. For convenience, edges such as $A \xrightarrow{c:x} C$ and $C \xrightarrow{C:-y} A$ may be notated as $(A, c{:}x, C)$ and $(C, C{:}{-y}, A)$, respectively.

An STNU is *dynamically controllable* (DC) if there exists a dynamic, real-time execution strategy that guarantees that all constraints in $\mathcal{C}$ will be satisfied no matter how the contingent durations turn out [13, 4]. A strategy is dynamic in that its execution decisions can react to observations of contingent executions, but with no advance knowledge of future events. Morris [10] proved that an STNU is DC if and only if it does not include any *semi-reducible* negative cycles (SRN cycles). (A path $\mathcal{P}$ is semi-reducible if certain constraint-propagation rules can be used to provide new edges that effectively *bypass* each occurrence of an LC edge in $\mathcal{P}$.) In 2014, Morris [11] presented the first $O(n^3)$-time DC-checking algorithm.[1] In 2018, Cairo *et al.* [1] presented their $O(mn + k^2 n + kn \log n)$-time `RUL`$^-$ DC-checking algorithm. Hunsberger and Posenato [6] subsequently presented a faster version, called `RUL2021`, that has the same worst-case complexity but achieves an order-of-magnitude speedup in practice by restricting the edges it inserts into the network during constraint propagation.

## 1.3 Flexible and Efficient Real-time Execution

Most DC-checking algorithms generate conditional *wait* constraints that must be satisfied by any valid execution strategy. Each wait is represented by a labeled edge of the form $W \xrightarrow{C:-w} A$, which may be notated as $(W, C{:}{-w}, A)$. (Despite the similar notation, a wait is distinguishable from the original UC edge since its source timepoint is *not* the contingent timepoint $C$.) Such a wait can be glossed as: "While $C$ remains unexecuted, $W$ must wait at least $w$ after $A$." Morris [11] defined an *Extended STNU* (ESTNU) to be an STNU augmented with such waits. Thus, the graph for an ESTNU includes a set $\mathcal{E}_{\text{ucg}}$ of generated wait edges. For convenience, we intentionally blur the distinction between an ESTNU and its graph.

[1] As is common in the literature, we use $n$ for the number of timepoints, $m$ for the number of ordinary constraints; and $k$ for the number of contingent links.

Morris then extended the notion of dispatchability to ESTNUs, defining it in terms of the ESTNU's STN projections. A *projection* of an ESTNU is the STN derived from assigning fixed values to the contingent durations. In any projection, each edge from the ESTNU projects onto an ordinary edge [12, 9]. For example, consider the contingent link $(A, 1, 10, C)$ and the projection where its duration $C - A$ equals 4. In that projection, the LC and UC edges, $(A, c{:}1, C)$ and $(C, C{:}{-10}, A)$, project onto the respective ordinary edges, $(A, 4, C)$ and $(C, -4, A)$, representing that $C - A = 4$. Meanwhile, the wait edges, $(W, C{:}{-7}, A)$ and $(V, C{:}{-3}, A)$, project onto $(W, -4, A)$ and $(V, C{:}{-3}, A)$, respectively, since the wait on $W$ expires when $C$ executes at $A + 4$, and the wait on $V$ is satisfied at time $A + 3$.

Morris defined an ESTNU to be dispatchable if all of its STN projections are dispatchable (as STNs). He then argued that a dispatchable ESTNU would necessarily provide a guarantee of flexible and efficient real-time execution. Hunsberger and Posenato [9] later:

1. formally defined a flexible and efficient real-time execution algorithm for ESTNUs, called RTE$^*$;
2. defined an ESTNU to be dispatchable if every run of RTE$^*$ necessarily satisfies all of the ESTNU's constraints; and
3. proved that an ESTNU satisfying their definition of dispatchability necessarily satisfies Morris' definition (i.e., all of its STN projections are STN-dispatchable).

The RTE$^*$ algorithm provides maximum flexibility during execution, unlike the *earliest-first* strategy used for non-dispatchable networks [5].

Most DC-checking algorithms do not generate dispatchable ESTNUs. However, Morris [11] argued that his $O(n^3)$-time DC-checking algorithm could be modified, without impacting its complexity, to generate a dispatchable output. In 2023, Hunsberger and Posenato [7] presented a faster, $O(mn + kn^2 + n^2 \log n)$-time ESTNU-dispatchability algorithm called $\texttt{FD}_{\texttt{STNU}}$. However, neither of these algorithms provides any guarantees about the number of edges in the dispatchable output. Since the number of edges in the network directly impacts the real-time computations required to execute the network, it is important to minimize that number. Hunsberger and Posenato [8] subsequently presented the first ESTNU-dispatchability algorithm, called $\texttt{minDisp}_{\texttt{ESTNU}}$, that, in $O(kn^3)$ time, generates an equivalent dispatchable ESTNU *having a minimal number of edges.* To date, it is the only such algorithm. The main contribution of this paper is to modify $\texttt{minDisp}_{\texttt{ESTNU}}$ so that it solves the same problem in $O(n^3)$-time, an order of magnitude faster, especially since it is common that $k = O(n)$, meaning the reduction in complexity is effectively from $O(n^4)$ to $O(n^3)$.

## 2 Overview of the Existing $\texttt{minDisp}_{\texttt{ESTNU}}$ Algorithm

The $\texttt{minDisp}_{\texttt{ESTNU}}$ algorithm [8] takes a dispatchable ESTNU $\mathcal{E} = (\mathcal{T}, \mathcal{E}_{\text{o}}, \mathcal{E}_{\text{lc}}, \mathcal{E}_{\text{uc}}, \mathcal{E}_{\text{ucg}})$ as its only input and generates as its output an equivalent dispatchable ESTNU having a minimal number of edges. (Such an ESTNU is called a $\mu$ESTNU for $\mathcal{S}$.) It has four steps:

1. Compute the set $\mathcal{E}_o^{\text{si}}$ of so-called *stand-in* edges: ordinary edges that are entailed by various combinations ordinary, LC, UC, and wait edges from the ESTNU.
2. Apply the STN-dispatchability algorithm from Tsamardinos *et al.* [16] to the resulting set of ordinary edges, thereby generating a dispatchable STN subgraph, $(\mathcal{T}, \mathcal{E}_{\text{o}}^*)$.
3. Let $\hat{\mathcal{E}}_{\text{o}}^* = \mathcal{E}_{\text{o}}^* \backslash \mathcal{E}_o^{\text{si}}$ be the result of removing any remaining stand-in edges from $\mathcal{E}_{\text{o}}^*$.
4. Compute the set of wait edges that are not needed for dispatchability and remove them from $\mathcal{E}_{\text{ucg}}$; call the resulting set $\hat{\mathcal{E}}_{\text{ucg}}$; then return the $\mu$ESTNU $(\mathcal{T}, \hat{\mathcal{E}}_{\text{o}}^*, \mathcal{E}_{\text{lc}}, \mathcal{E}_{\text{uc}}, \hat{\mathcal{E}}_{\text{ucg}})$.

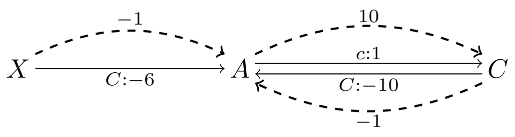


**Figure 1** (Dashed) stand-in edges entailed by individual labeled edges.

The worst-case time complexity of the $\texttt{minDisp}_{\texttt{ESTNU}}$ algorithm is dominated by the first step: finding the set $\mathcal{E}_o^{\text{si}}$ of so-called *stand-in* edges. Therefore, our new, faster algorithm modifies only that step, achieving an order-of-magnitude reduction in the overall worst-case time complexity. The rest of this section gives an overview of Step 1 of the existing $\texttt{minDisp}_{\texttt{ESTNU}}$ algorithm, as implemented by its `genStandIns` helper algorithm.

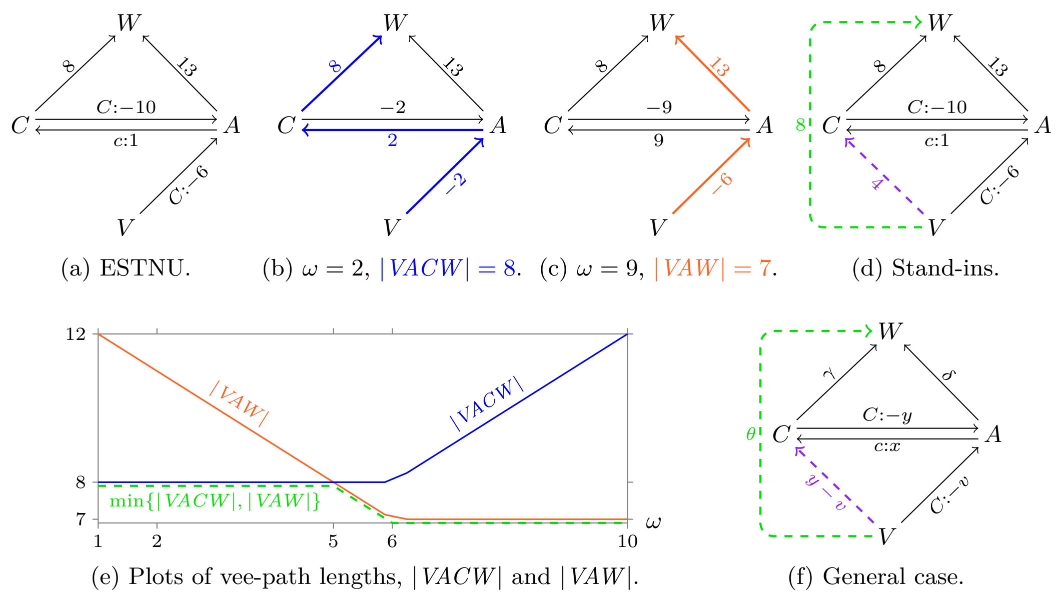
## 2.1 Generating Stand-in Edges

Following Morris [11, 12], an ESTNU is dispatchable if all of its STN projections are dispatchable (as STNs). That, in turn, requires that in each STN projection, each pair of timepoints $V$ and $W$ that are connected by a path be connected by a *shortest vee-path* (i.e., a path comprising zero or more negative edges followed by zero or more non-negative edges). A key insight behind the $\texttt{minDisp}_{\texttt{ESTNU}}$ algorithm is that in different projections, the shortest vee-paths from $V$ to $W$ may take different routes and may have different lengths.

Before addressing more complex cases, `genStandIns` generates stand-in edges entailed by individual labeled edges. For example, given a contingent link $(A, x, y, C)$, the LC edge $(A, c{:}x, C)$ entails a stand-in edge $(A, y, C)$ because in any projection where $\omega = C - A \in [x, y]$, the LC edge projects onto the ordinary edge $(A, \omega, C)$, whose length is $\omega \le y$. Similarly, the UC edge $(C, C{:}{-y}, A)$ entails a stand-in edge $(C, -x, A)$ since in any projection the UC edge projects onto the ordinary edge $(C, -\omega, A)$, whose length is $-\omega \le -x$. Finally, a wait edge $(V, C{:}{-v}, A)$, where $-v < -x$, projects onto the ordinary edge $(V, \max\{-\omega, -v\}, A)$ and hence entails a stand-in edge $(V, -x, A)$, since $-\omega \le -x$ and $-v < -x$.[2] Figure 1 shows an example of the stand-in edges entailed by individual labeled edges.

The most computationally costly part of the `genStandIns` algorithm is its computation of stand-in edges entailed by different combinations of ESTNU edges. For example, consider the ESTNU in Figure 2a, commonly referred to as a *diamond structure*. In the projection where $\omega = C - A = 2$, the projected path $VACW$, shown in blue in Figure 2b, is the shortest vee-path from $V$ to $W$: its length is 8. But in the projection where $\omega = C - A = 9$, the projected path $VAW$, shown in orange in Figure 2c, is the shortest vee-path from $V$ to $W$: its length is 7. The plots of the lengths, $|VACW|$ and $|VAW|$, in Figure 2e, show that across *all* projections the *maximum* length of the *shortest* vee-path from $V$ to $W$, indicated by the dashed green line, is 8. In other words, the combination of edges in the diamond structure entails the stand-in edge $(V, 8, W)$, shown as dashed and green in Figure 2d. Since the constraint, $W - V \le 8$, must be satisfied in all projections, it must also be satisfied by any dynamic execution strategy for the ESTNU. Similarly, the path $VAC$ satisfies $|VAC| = \max\{-\omega, -6\} + \omega = \max\{0, \omega - 6\} \le 4$, for all $\omega \in [1, 10]$. Thus, that path entails the stand-in edge $(V, 4, C)$, shown as dashed and purple in Figure 2d. Like all stand-in edges, it must be satisfied by any dynamic execution strategy. The purpose of

[2] As a first step, `genStandIns` replaces *weak* waits (i.e., those where $-v \ge -x$) by ordinary edges and adjusts *misleading* waits (i.e., those where $-v < -y$). But those details are not important for this paper.



(a) ESTNU. (b) $\omega = 2$, $|VACW| = 8$. (c) $\omega = 9$, $|VAW| = 7$. (d) Stand-ins.

(e) Plots of vee-path lengths, $|VACW|$ and $|VAW|$. (f) General case.

**Figure 2** (a) Sample ESTNU, (b) and (c) two of its projections with (colored) shortest *vee-paths*, (d) entailed (dashed) stand-in edges, (e) plots of vee-path lengths, and (f) the general case.

the `genStandIns` helper algorithm is to make all such constraints temporarily explicit so that Step 2 of $\texttt{minDisp}_{\texttt{ESTNU}}$ can determine which ordinary edges can be removed without threatening the dispatchability of the ESTNU.

Each iteration of the `genStandIns` algorithm's main loop explores $O(n^2k)$ diamond structures ($n$ choices for $V$, $n$ choices for $W$, and $k$ choices for the contingent link), as illustrated in Figure 2f, where the distances $\delta$ and $\gamma$ are provided by the all-pairs shortest-paths (APSP) matrix for the ordinary edges in the ESTNU. (The APSP matrix for the ordinary edges is commonly called the *distance matrix,* denoted by $\mathcal{D}$.) The lengths of the alternative vee-paths, $VACW$ and $VAW$, are given by $|VACW| = \max\{-\omega, -v\} + \omega + \gamma$ and $|VAW| = \max\{-\omega, -v\} + \delta$. Their intersection occurs where $\omega = \delta - \gamma$. If that value falls within the interval $(x, y)$, it is not hard to show that the maximum length of any shortest vee-path from $V$ to $W$ across *all* projections is $\theta = \max\{\gamma, \delta - v\}$, represented by the stand-in edge $(V, \theta, W)$, shown as dashed in Figure 2f. The other stand-in edge $(V, y - v, C)$ derives from the two-edge path, $VAC$, whose length in the projection where $\omega = C - A$ is given by: $|VAC| = \max\{-\omega, -v\} + \omega = \max\{0, \omega - v\} \leq y - v$. After exploring all such diamond structures, Johnson's algorithm [2] is called to update the APSP matrix.

## 2.2 Stand-in edges arising from nested diamond structures

Because the distances involved in the analysis of diamond structures depend on shortest paths in the subgraph of ordinary edges (e.g., $\gamma = \mathcal{D}(C, W)$ and $\delta = \mathcal{D}(A, W)$ in Figure 2f), which can be affected by inserting (ordinary) stand-in edges into the ESTNU, it follows that stand-in edges can derive from *nested* diamond structures, for example, as illustrated in Figure 3. That figure 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: $\max\{-\omega_2, -6\} + 9 = \max\{9 - \omega_2, 3\} \leq 3$.

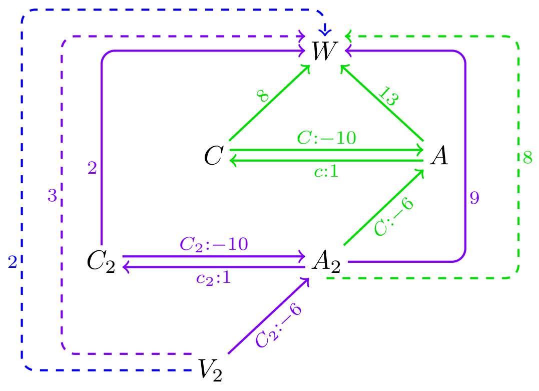


**Figure 3** Deriving stand-in edges from nested diamond structures.

Next, since the green diamond is isomorphic to the diamond from Figure 2d, 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 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.

Hunsberger and Posenato [8] proved that it suffices to explore nested diamond structures up to a maximum depth of $k$. Thus, the `genStandIns` algorithm does up to $k$ iterations of its main loop. Since each iteration ends by calling Johnson's algorithm on up to $n^2$ edges, the overall complexity of `genStandIns` is $O(kn^3)$.

## 3 Speeding up the $\texttt{minDisp}_{\texttt{ESTNU}}$ Algorithm

The complexity of the $\texttt{minDisp}_{\texttt{ESTNU}}$ algorithm is driven by the $O(kn^3)$-time complexity of `genStandIns`. Our modification of $\texttt{minDisp}_{\texttt{ESTNU}}$ replaces `genStandIns` with `newGenStandIns`, which, taking a more focused and efficient approach to dealing with nested diamond structures, works in $O(n^3)$ time. Since $k = O(n)$ is common in applications (e.g., $k \approx n/10$ in some benchmarks [15]), the reduction in worst-case time-complexity is effectively from $O(n^4)$ to $O(n^3)$.

### 3.1 Stand-in Edges Derived from Nested Diamond Structures

Figure 4 illustrates the nested relationship between an inner diamond $D_i$ (involving timepoints $V_i, A_i, C_i$ and $W_i$, shaded dark gray) and an outer diamond $D_j$ (involving timepoints $V_j, A_j, C_j$ and $W_j$, shaded light gray), where the arrows labeled by $a, b, \delta_i, \gamma_i$ and $\gamma_j$ represent ordinary edges or paths, and the dashed arrows represent the stand-in edges $(V_i, \tau_i, W_i)$ and $(V_j, \tau_j, W_j)$ entailed by the diamonds.[3] Lemma 1, below, ensures that in any such nesting, there must be a path from $A_j$ to $A_i$ that comprises zero or more negative ordinary edges followed by one (negative) wait edge, which for convenience we call a *negOrdWait* path. This implies that the activation timepoints involved in nested structures can be put into a strict partial order which, in turn, implies that generating the stand-in zedges associated

[3] Hunsberger and Posenato [8] proved that when considering vee-paths from $A_j$ to $W_j$, the only relevant nesting of diamonds occurs if the inner diamond $D_i$ resides along the path from $A_j$ to $W_j$ in the outer diamond $D_j$, as shown in the figure. Since the inner diamond begins with a negative wait edge, any path from $A_j$ to $W_j$ that included $D_i$ between $C_j$ and $W_j$ could not be a vee-path.

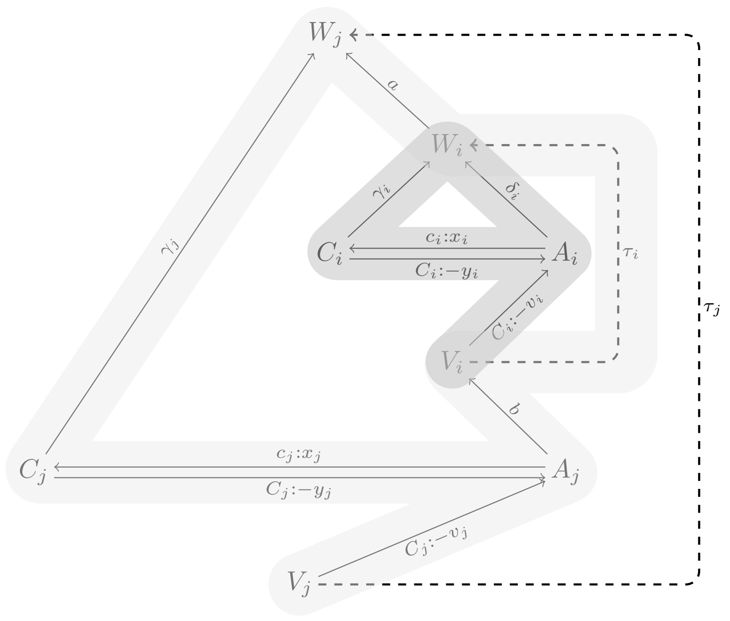


**Figure 4** Nested diamond structures (one shaded light, one shaded dark) considered in Lemma 1.

with nested diamonds can be done in just one pass, instead of the $k$ passes through the main loop of `genStandIns`. Furthermore, to determine the length of the stand-in edge from $V_j$ to any $W_j$, taking advantage of the nesting of $D_i$ within $D_j$, it suffices to know the length of the shortest ordinary path from $A_j$ to $W_j$. (Recall that the length of the entailed stand-in edge depends only on the values of $\mathcal{D}(C_j, W_j)$, $\mathcal{D}(A_j, W_j)$ and $-v_j$.) In other words, when generating stand-in edges derived from diamonds involving the labeled edges $(A_i, c_i{:}x_i, C_i)$ and $(C_i, C_i{:}-y_i, A_i)$, it is not necessary to find all ordinary distances affected by those stand-in edges (which is what the existing `genStandIns` algorithm uses Johnson's algorithm to do – *on each of up to $k$ passes*); instead, it suffices to focus on the distances of ordinary paths emanating from $A_j$ that are affected by those stand-in edges. In the case of $A_j$ shown in the figure, it suffices to record distances of the form, $\mathcal{D}(A_j, W_i) = b + \tau_i$, resulting from new stand-in edges. Crucially, all of these distances correspond to paths emanating from a single source, $A_j$. After exploring all inner diamonds $D_i$ and recording the new distances, $\mathcal{D}(A_j, W_i)$, then *all* values $\mathcal{D}(A_j, \cdot)$ can be updated using Dijkstra's single-source shortest-paths algorithm, guided by a potential function [2]. These observations enable the `newGenStandIns` algorithm, presented later in this section, to call Dijkstra's algorithm $k$ times, instead of calling Johnson's algorithm $k$ times, leading to an order-of-magnitude reduction in worst-case time complexity, from $O(kn^3)$ down to $O(n^3)$.

▶ **Lemma 1.** *Let $\mathcal{S}$ be any dispatchable ESTNU. Suppose that $E$ is a stand-in edge derived from nested diamond structures in which the diamond structure $D_i$ associated with the contingent link $(A_i, x_i, y_i, C_i)$ is nested directly inside the diamond structure $D_j$ associated with the contingent link $(A_j, x_j, y_j, C_j)$. Furthermore, suppose that the labeled edges from these contingent links are* needed *for $E$ (i.e., without their labeled edges, $E$ would not be entailed by the remaining edges in $\mathcal{S}$). Then there must be a path from $A_j$ to $A_i$ in $\mathcal{S}$ that consists of zero or more negative ordinary edges, followed by a single wait edge of the form $(V_i, C_i{:}-v_i, A_i)$ (i.e., a* negOrdWait *path).*

**Proof.** Suppose that $E$ is the stand-in edge $(V_j, \tau_j, W_j)$. Since the labeled edges from these contingent links are needed for $E$, it follows that in at least one STN projection, the shortest vee-path from $V_j$ to $W_j$ must include the path from $A_j$ to $V_i$ to $A_i$. Since any subpath of a vee-path is also a vee-path and the wait edge $(V_i, C_i{:}{-v_i}, A_i)$ has negative length, it follows that all of the ordinary edges represented in the figure by $(A_j, b, V_i)$ must be negative. ◀

Given Lemma 1, the activation timepoints participating in a nested diamond structure must be linked by a chain of *negOrdWait* paths. In addition, for a DC STNU, there can be no cycles of such paths because they would constitute a negative cycle in the OU-graph, i.e., $\mathcal{G}_{ou} = (\mathcal{T}, \mathcal{E}_{\mathrm{o}} \cup \mathcal{E}_{\mathrm{uc}} \cup \mathcal{E}_{\mathrm{ucg}})$, the graph containing all the original and derived edges but the lower-case ones. However, a single activation timepoint may participate in multiple nested structures. Hence, the set of all *negOrdWait* paths among the activation timepoints necessarily forms a strict partial order (equivalently, a forest of one or more directed acyclic graphs in the OU-graph).

For each pair of activation timepoints, $A_j$ and $A_i$, for which there is a *negOrdWait* path from $A_j$ to $A_i$, we say that $A_j$ is a *parent* of $A_i$ and that $A_i$ is a *child* of $A_j$. The relevant information for determining the stand-in edges emanating from $A_j$ and passing through a diamond structure involving labeled edges from $(A_i, x_i, y_i, C_i)$ is: (1) $\ell$, the (negative) length of the *negOrdWait* path from $A_j$ to $A_i$; and (2) $-v_i$, the (negative) length of the wait edge, $(V_i, C_i{:}{-v_i}, A_i)$, terminating that *negOrdWait* path. These lengths are shown in Figure 4, where $b = \ell + v_i$ is the length of the prefix of the *negOrdWait* path that includes only the ordinary edges (i.e., everything except the terminal wait edge). Then, as shown by Hunsberger and Posenato [8], for any timepoint $W_i \in \mathcal{T} \backslash \{A_i, C_i, A_j, C_j\}$, the length of the potential stand-in edge from $A_j$ to $W_i$ is given by $b + \max\{\gamma_i, \delta_i - v_i\} = \max\{b + \gamma_i, \ell + \delta_i\}$, where $\gamma_i = \mathcal{D}(C_i, W_i)$ and $\delta_i = \mathcal{D}(A_i, W_i)$, also shown in the figure. Then, for any $W_j$, the ordinary distance $\mathcal{D}(A_j, W_j)$ affected by such a stand-in edge can be determined by the previously mentioned call to Dijkstra's algorithm, guided by a potential function.

## 3.2 The `getPCinfo` (get parent/child info) Algorithm

The `getPCinfo` algorithm (Algorithm 1) efficiently computes the relevant parent/child information, returning a pair of vectors of hash tables, called *parent* and *child*. For each activation timepoint $A_i$, *parent*$[A_i]$ is a hash table containing entries where some $A_j$ is the key and $(\ell, -v_i)$ is the value (i.e., $A_j$ is the parent, $\ell$ is the length of the *negOrdWait* path from $A_j$ to $A_i$, and $-v_i$ is the length of its terminating wait edge). Similarly, for each activation timepoint $A_j$, *child*$[A_j]$ is a hash table containing entries linking some child $A_i$ to the corresponding pair $(\ell, -v_i)$, where $\ell$ is the length of the *negOrdWait* path from $A_j$ to $A_i$, and $-v_i$ is the length of its terminal wait edge.

An important factor is that if two *negOrdWait* paths from $A_j$ to $A_i$ have the same length, but one has a stronger (i.e., more negative) terminating wait edge, then the *negOrdWait* path terminated by the *weaker* wait *dominates* the one with the stronger wait because in any projection the projected length of the one with the weaker wait will be shorter than (or the same as) that of the one with the stronger wait. For example, if $\ell$ is the length of two *negOrdWait* paths from $A_j$ to $A_i$, but $-v_1 > -v_2$, where the corresponding terminal wait edges are $(V_1, C_i{:}{-v_1}, A_i)$ and $(V_2, C_i{:}{-v_2}, A_i)$, then $|A_j\, V_1\, A_i| = (\ell + v_1) + \max\{-\omega, -v_1\} = \max\{\ell + v_1 - \omega, \ell\} \leq \max\{\ell + v_2 - \omega, \ell\} = (\ell + v_2) + \max\{-\omega, -v_2\} = |A_j\, V_2\, A_i|$. Another important factor involves *negOrd* paths (i.e., paths comprising solely negative ordinary edges). If a *negOrd* path has the same length as a *negOrdWait* path, then the *negOrd* path dominates the *negOrdWait* path since in every projection the length of the *negOrd* path will be the same as or shorter than the length of the projected *negOrdWait* path.

**Algorithm 1** `getPCinfo`: find *negOrdWait* paths between pairs of activation timepoints.

```
Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E}_o, \mathcal{E}_{lc}, \mathcal{E}_{uc}, \mathcal{E}_{ucg})$, an ESTNU graph
Output: $(parent, child)$, where $parent$ and $child$ are $k$-vectors of hash tables signaling the
        presence of $negOrdWait$ paths between pairs of activation timepoints
$f :=$ bellmanFord($\mathcal{G}_{ou}$)        // A potential function for $\mathcal{G}_{ou} = (\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_{uc} \cup \mathcal{E}_{ucg})$
$parent := (\emptyset, \ldots, \emptyset)$
$child := (\emptyset, \ldots, \emptyset)$        // $k$-vectors of hash tables
foreach $(A, x, y, C) \in \mathcal{L}$ do        // Back-propagate from $A$ along negOrdWait paths
    $negLen := (\infty, \ldots, \infty)$   // An $n$-vector of accum. lengths of $negOrdWait$ paths ending in $A$
    $negWait := (\perp, \ldots, \perp)$   // An $n$-vector of corresp. neg. wait values (or $\perp$ for ord paths)
    // Initialize min priority queue $\mathcal{Q}$ with entries for negative ord and wait edges incoming to $A$
    //   Element = $U$, a timepoint
    //   Key = Non-negative accumulated length adjusted by potential function, $f$
    $\mathcal{Q} :=$ new priority queue
    foreach $(U, \delta, A)$ with $\delta < 0$ do        // Negative ordinary edges incoming to $A$
        $\mathcal{Q}.insert(U, \delta - f(A) + f(U))$        // $f(A) - f(U) \leq \delta \iff \delta - f(A) + f(U) \geq 0$
        $negLen[U] := \delta$
    foreach $(V, C{:}{-v}, A) \in \mathcal{E}_{ucg}$ do        // (Negative) wait edges incoming to $A$
        $\mathcal{Q}.insert(V, -v - f(A) + f(V))$        // $f(A) - f(V) \leq -v \iff -v - f(A) + f(V) \geq 0$
        $negLen[V] := -v$;  $negWait[V] := -v$
    // Use back-propagation to find shortest $negOrd$ or $negOrdWait$ paths terminating at $A$
    while $\neg\mathcal{Q}.empty()$ do
        $U := \mathcal{Q}.extractMin()$
        if $U = A'$ is an activation timepoint and $negWait[A'] \neq \perp$ then
            // Record $negOrdWait$ path found from $A'$ to $A$
            $parent[A].insert(A', (negLen[A'], negWait[A']))$
            $child[A'].insert(A, (negLen[A'], negWait[A']))$
        // Continue back-propagating along negative ordinary edges
        foreach $(V, v, U) \in \mathcal{E}_o \mid v < 0$ do
            $newLen := v + negLen[U]$
            if $newLen < negLen[V]$ or $((newLen == negLen[V])$ and
             $((negWait[U] == \perp)$ or $(negWait[U] > negWait[V])))$ then
                // Record new shortest $negOrd$ or $negOrdWait$ path from $V$ to $A$ (via $U$)
                if $negLen[V] == \infty$ then  $\mathcal{Q}.insert(V, newLen - f(A) + f(V))$
                else  $\mathcal{Q}.decreaseKey(V, newLen - f(A) + f(V))$
                $negLen[V] := newLen$
                $negWait[V] := negWait[U]$
return $(parent, child)$        // Return the vectors of parent/child hash tables
```

At Line 1, `getPCinfo` calls the Bellman-Ford algorithm [2] to generate a solution to the OU-graph that will be used as a potential function to guide the traversal of negOrd and negOrdWait paths. Line 2 initializes the *parent* and *child* vectors of hash tables.

Each iteration of the `for` loop at Lines 4–25 processes one activation timepoint $A$, looking for shortest *negOrd* or *negOrdWait* paths from $A$ *backward* to other activation timepoints. Lines 5–6 initialize the *negLen* and *negWait* vectors. For each $X$, $negLen[X]$ specifies the length of the shortest *negOrd* or *negOrdWait* path from $X$ to $A$ that has been found so far (or $\infty$). If a shortest *negOrdWait* path from $X$ to $A$ has been found that is not dominated by a *negOrd* path, then $negWait[X]$ specifies the length of its terminating wait edge.

Lines 7–13 initialize a min priority queue [2] to include an entry for each negative ordinary edge and each wait edge incoming to $A$. Like in Johnson's algorithm, the potential function $f$ is used to adjust the distances in the OU-graph to be non-negative to enable the use of Dijkstra's algorithm to guide the exploration of *negOrd* and *negOrdWait* paths.

Each iteration of the `while` loop (Lines 14–25) pops a timepoint $U$ off the queue. If $U$ happens to be an activation timepoint $A'$ for which an undominated *negOrdWait* path has been found, then entries linking $A$ (the child) to $A'$ (the parent) are inserted into the relevant hash tables (Lines 16–18). Next, back-propagation along negative ordinary edges continues at Lines 19–25. The complicated `if` condition at Line 21 covers cases where a new shortest *negOrd* or *negOrdWait* path from $V$ to $A$ (via $U$) has been found. First, if $\mathit{newLen} < \mathit{negLen}[V]$ (which includes $\mathit{negLen}[V] = \infty$), then the path via $U$ is a new shortest path. Second, if $\mathit{newLen} = \mathit{negLen}[V]$, then the path via $U$ dominates a pre-existing path from $V$ to $A$ if: (1) the path via $U$ is a *negOrd* path (whence $\mathit{negWait}[U] = \perp$); or (2) the wait terminating the path via $U$ is weaker than the terminal wait in the pre-existing path (i.e., $\mathit{negWait}[U] > \mathit{negWait}[V]$). In any of these cases, the values of $\mathit{negLen}[V]$ and $\mathit{negWait}[V]$ are updated, and $V$ is either newly inserted into the queue or its key is updated (Lines 22–25). After the main `for` loop is completed, the *parent* and *child* vectors of hash tables are returned at Line 26.

## 3.3 The `newGenStandIns` Algorithm

The section presents our `newGenStandIns` algorithm (Algorithm 2). It uses the *parent* and *child* hash tables computed by `getPCinfo` to more efficiently generate all of the stand-in edges arising from nested diamond structures. Its time-complexity is $O(n^3)$, an order-of-magnitude improvement over the $O(kn^3)$-time complexity of `genStandIns`.

For simplicity, we assume that all stand-in edges entailed by *individual* labeled edges have already been computed and have been passed as an input $\mathcal{E}_{\text{isi}}$ into `newGenStandIns`.

At Line 1, `newGenStandIns` calls the Bellman-Ford algorithm on the subgraph of ordinary edges which will be used as a potential function to enable the use of Dijkstra's single-source shortest-paths algorithm to update distance-matrix entries. At Line 2, $\mathcal{E}_{\text{o}}^t$ is initialized; it will accumulate changes to $\mathcal{D}(A_j, \cdot)$ values, stored as *temporary edges,* that are derived directly from nested stand-in edges. Next, at Lines 3–7, the list, *readyToGo*, of activation timepoints that are ready to process is initialized. Since the activation timepoints form a strict partial order, this list is initially populated by those having no children. The vector, *numUnprocd*, keeps track of how many unprocessed children each activation timepoint has. Later on, as each activation timepoint is processed, its parent's entry in *numUnprocd* will be decremented.

Each iteration of the `while` loop (Lines 8–28) pops one activation timepoint $A_j$ off the *readyToGo* list and, at Lines 12–19, for each child $A_i$ and each timepoint $W_i$, explores diamond structures involving the labeled edges from the contingent link $(A_i, x_i, y_i, C_i)$, to determine whether the distance $\mathcal{D}(A_j, W_i)$ can be affected by a nested diamond. (Recall Figure 3.) Instead of explicitly dealing with the wait edge $(V_i, C_i{:}{-v_i}, A_i)$ shown in the figure, `newGenStandIns` uses the $\ell_i$ and $-v_i$ values retrieved from the $\mathit{child}[A_j]$ hash table at Line 12 (where $b$ in the figure equals $\ell_i + v_i$), along with the distances, $\gamma_i = \mathcal{D}(C_i, W)$ and $\delta_i = \mathcal{D}(A_i, W_i)$, obtained from the distance matrix at Line 14. This information is sufficient to determine whether the paths $V_iA_iC_iW_i$ and $V_iA_iW_i$ combine to entail a new stand-in edge, $(V_i, \tau_i, W_i)$, where $\tau_i = \max\{\gamma_i, \delta_i - v_i\}$. In particular, as in `genStandIns`, $\omega_i = \delta_i - \gamma_i$ (at Line 15) specifies the projection where $|\,V_iA_iC_iW_i| = |\,V_iA_iW_i|$; and a new stand-in edge from $V_i$ to $W_i$ is entailed if $\omega_i \in (x_i, y_i)$ and if that new stand-in edge is at least as strong as any existing ordinary path from $V_i$ to $W_i$. However, here, the goal is not to generate

**Algorithm 2** `newGenStandIns`: Compute the stand-in edges arising from nested diamonds.

```
Input: $(\mathcal{T}, \mathcal{E}_{\rm o}, \mathcal{E}_{\rm lc}, \mathcal{E}_{\rm uc}, \mathcal{E}_{\rm ucg})$, dispatchable ESTNU; *parent*, *child*, vectors of hash tables
       computed by getPCinfo; $\mathcal{D}$, distance matrix for $\mathcal{G}_o = (\mathcal{T}, \mathcal{E}_{\rm o})$; $\mathcal{E}_{\rm isi} \subseteq \mathcal{E}_{\rm o}$, stand-in edges
       entailed by *individual* labeled edges
Output: $\mathcal{E}_{\rm si}$, the set of *all* stand-in edges (including $\mathcal{E}_{\rm isi}$); and $\mathcal{D}$, the updated distance matrix.
1  $f :=$ bellmanFord($\mathcal{G}_o$)   // Initialize a potential function $f$ on the ordinary subgraph $\mathcal{G}_o$
2  $\mathcal{E}_{\rm o}^t := \emptyset$   // Used to collect all temporary (ordinary) edges
3  $readyToGo := \emptyset$   // A list of activation timepoints ready for processing
4  $numUnprocd := (0, \dots, 0)$   // For each activ'n. timepoint, the num of its unprocessed children
5  foreach $(A, x, y, C) \in \mathcal{L}$ do
6      $numUnprocd[A] := child[A].count()$   // Fetch the number of $A$'s children
7      if $numUnprocd[A] == 0$ then $readyToGo.push(A)$   // If no children, then ready to process
8  while $readyToGo \neq \emptyset$ do
9      $A_j := readyToGo.pop()$   // Contingent link for $A_j$ is $(A_j, x_j, y_j, C_j)$
10     $anyChange := \bot$
11     $newLengths :=$ empty hash table   // For collecting new $\mathcal{D}(A_j, \cdot)$ values
12     foreach $(A_i, (\ell_i, -v_i)) \in child[A_j]$ do   // Contingent link for $A_i$ is $(A_i, x_i, y_i, C_i)$
13         foreach $W_i \in \mathcal{T} \backslash \{A_i, C_i, A_j, C_j\}$ do
14             $\gamma_i = \mathcal{D}(C_i, W_i)$;  $\delta_i = \mathcal{D}(A_i, W_i)$;  $\omega_i := \delta_i - \gamma_i$
15             if $\omega_i \in (x_i, y_i)$ then   // $\omega_i$ specifies proj'n. where max shortest vee-path occurs
16                 $newVal := \max\{\ell_i + v_i + \gamma_i, \ell_i + \delta_i\}$ // Length of potential new $\mathcal{D}(A_j, W_i)$ value
17                 if $newVal < \mathcal{D}(A_j, W_i)$ then
18                     $newLengths.insert(W_i, newVal)$   // Record new $\mathcal{D}(A_j, W_i)$ value
19                     $anyChange := \top$
20     if $anyChange == \top$ then   // Need to update potential function and $\mathcal{D}(A_j, \cdot)$ values
21         $\mathcal{E}_{\rm o}^+ := \emptyset$   // Collect set of changed $\mathcal{D}(A_j, \cdot)$ values as temporary edges
22         foreach $(W_i, newVal) \in newLengths$ do $\mathcal{E}_{\rm o}^+ := \mathcal{E}_{\rm o}^+ \cup \{(A_j, newVal, W_i)\}$
23         $f :=$ updatePotFn($(\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{\rm o}^+), f$)   // Update pot'l. fn. to accommodate temp edges
24         $\mathcal{D}(A_j, \cdot) :=$ dijkstra($A_j, \mathcal{E}_{\rm o} \cup \mathcal{E}_{\rm o}^+, f$)   // Update $\mathcal{D}(A_j, \cdot)$ values for next iteration
25         $\mathcal{E}_{\rm o}^t := \mathcal{E}_{\rm o}^t \cup \mathcal{E}_{\rm o}^+$   // Accumulate temp edges RE: $A_j$ in global set $\mathcal{E}_{\rm o}^t$
26     foreach $A \in parent[A_j]$ do   // Update info for $A_j$'s parents now that $A_j$ is done
27         $numUnprocd[A] := numUnprocd[A] - 1$
28         if $numUnprocd[A] == 0$ then $readyToGo.push(A)$
   // Fully updated $\mathcal{D}$ ensures that *one* iteration of genStandIns will generate *all* stand-in edges
29 $\mathcal{D} :=$ johnson($\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{\rm o}^t$)   // After this, temp edges are discarded
30 $\mathcal{E}_{\rm si} :=$ genStandInsOnce($(\mathcal{T}, \mathcal{E}_{\rm o}, \mathcal{E}_{\rm lc}, \mathcal{E}_{\rm uc}, \mathcal{E}_{\rm ucg}), \mathcal{E}_{\rm isi}, \mathcal{D}$)
31 $\mathcal{D} :=$ johnson($\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{\rm si}$)   // Final update of $\mathcal{D}$ to accommodate the generated stand-in edges
32 return $(\mathcal{E}_{\rm si}, \mathcal{D})$
```

that stand-in edge, but instead to provide the $\mathcal{D}(A_j, W_i)$ value affected by it. Therefore, the only information accumulated in the *newLengths* hash table is the pair $(W_i, newVal)$, where $newVal = b + \tau_i = \ell_i + v_i + \tau_i = \max\{\ell_i + v_i + \gamma_i, \ell_i + \delta_i\}$ (at Lines 16–18).

Afterward, at Line 20, if processing $A_j$ led to changes in any $\mathcal{D}(A_j, \cdot)$ values, then `newGenStandIns` collects all of the changes as a set $\mathcal{E}_{\rm o}^+$ of *temporary* edges (Lines 21–22) that it then uses to (1) incrementally update the potential function $f$ (at Line 23), and (2) propagate the new $\mathcal{D}(A_j, \cdot)$ values to update *all* affected $\mathcal{D}(A_j, \cdot)$ values (at Line 24). For updating the potential function, it calls the `updatePotFn`, which is a simplified version of the `UpdPF` algorithm from the `RUL2021` algorithm [6]; here, it explores paths emanating from $A_j$ as long as changes to the potential function are needed. For updating $\mathcal{D}(A_j, \cdot)$ values, it calls

**Algorithm 3** The `updatePotFn` function.

```
Input: $\mathcal{G}_o = (\mathcal{T}, \mathcal{E}_{\rm o})$, STN; $A$, timepoint; $h$, pot'l. fn. for $\mathcal{G}_o$, excluding edges emanating from $A$
Output: A pot'l. fn. $h'$ for $\mathcal{G}_o$ (including edges emanating from $A$); or $\perp$ if $\mathcal{G}_o$ is inconsistent
h' := copy-vector(h)
Q := new empty priority queue
Q.insert(A, 0)                                  // Initialize queue for forward propagation from A
while (!Q.empty()) do
    (V, key(V)) := Q.extractMinNode()
    foreach ((V, δ, W) ∈ E_o) do              // Propagate along ordinary edges emanating from V
        if (h'(W) > h'(V) + δ) then
            h'(W) := h'(V) + δ  // Update pot'l. fn. h' and insert W into Q or decrease its key
            if (Q.state(W) == notYetInQ) then  Q.insert(W, h(W) − h'(W))
            else  Q.decreaseKey(W, h(W) − h'(W))
return h'
```

Dijkstra's single-source shortest-paths algorithm using $A_j$ as the source and $f$ as a potential function to re-weight the edges to non-negative values. This use of Dijkstra is similar to its use in Johnson's algorithm [2]. Note that after these updates the temporary edges in $\mathcal{E}_{\rm o}^{+}$ are *not* inserted into the ESTNU graph, but they are accumulated in $\mathcal{E}_{\rm o}^{t}$ for later use at Line 25.

The processing of $A_j$ ends at Lines 26–28, where for each parent $A$ of $A_j$, the number of $A$'s unprocessed children is decremented by 1 and, if that number reaches 0, then $A$ is pushed onto the *readyToGo* list, indicating that it is ready for processing.

Once all activation timepoints have been processed, all distance values $\mathcal{D}(A_j, \cdot)$ needed to account for arbitrary nestings of diamond structures have been accumulated. All that remains is to *use* these values to generate all of the stand-in edges. For example, suppose that the diamond formed by $V_j, A_j, C_j$ and $W_j$ from Figure 3 is the *outermost* diamond in a nested sequence that entails a stand-in edge of the form, $(V_j, \tau_j, W_j)$. Then the resulting $\mathcal{D}(A_j, W_j)$ value, determined by the inner levels of nesting, was computed when $A_j$ was processed by the `while` loop at Lines 8–19. But the stand-in edge $(V_j, \tau_j, W_j)$ has not yet been generated. However, given all of the $\mathcal{D}(A_j, \cdot)$ values computed so far (for all $A_j$), generating *all* such stand-in edges, including those that are *not* involved in any nesting, can be accomplished by an $O(kn^2)$-time exploration of diamond structures involving any timepoints, $V, A, C, W$, where $A$ and $C$ are timepoints associated with a contingent link $(A, x, y, C)$, and $V$ and $W$ are any timepoints other than $A$ or $C$. This is precisely what a *single iteration* of the `for` loop at Lines 13-27 of `genStandIns` does. Here, it is called `genStandInsOnce`, at Line 30. Afterward, at Line 31, a final call to Johnson's algorithm computes the full distance matrix to accommodate all of the new stand-in edges, including those in $\mathcal{E}_{\rm isi}$ passed in as an input.

## 3.4 Complexity of `newGenStandIns`

Our modification of the `minDisp`$_{\texttt{ESTNU}}$ algorithm replaces the `genStandIns` helper by the `newGenStandIns` algorithm presented above. The complexity of `newGenStandIns` is determined as follows. Its $k$ calls of Dijkstra's algorithm on at most $m + nk$ edges cost $O(mk + nk^2 + kn \log n)$ time. Its $k$ calls of the `updatePotFn` function similarly require $O(mk + nk^2 + kn \log n)$ time. The call to `genStandInsOnce`, as reported by Hunsberger and Posenato [8], requires $O(kn^2)$ time ($n$ choices for $V$, $n$ choices for $W$, and $k$ choices for $(A, x, y, C)$). The most costly computation, however, is the last one: the call to Johnson's algorithm on at most $m = n^2$ edges costs $O(n^3)$ time. Therefore, the overall complexity of `newGenStandIns` is $O(n^3)$. This is an order-of-magnitude reduction compared to the $O(kn^3)$ complexity of `genStandIns`, especially since, for applications, $k = O(n)$ (e.g., $k \approx n/10$ in some benchmarks [15]), implying an effective reduction from $O(n^4)$ to $O(n^3)$.

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

## 4 Conclusions

Generating an equivalent dispatchable ESTNU having a minimal number of edges is an important problem for applications involving actions with uncertain but bounded durations. The number of edges in the dispatchable network is important because it directly impacts the real-time computations that are necessary when executing the network. Therefore, for time-sensitive applications it is important to generate an equivalent dispatchable ESTNU having a minimal number of edges, called a $\mu$ESTNU. This paper modified the only existing algorithm for generating a $\mu$ESTNU, making it an order-of-magnitude faster. It reduced the worst-case time-complexity from $O(kn^3)$ to $O(n^3)$ which, given that in typical applications $k = O(n)$, implies an effective reduction from $O(n^4)$ to $O(n^3)$.

## References

1 Massimo Cairo, Luke Hunsberger, and Romeo Rizzi. Faster Dynamic Controllablity Checking for Simple Temporal Networks with Uncertainty. In *25th International Symposium on Temporal Representation and Reasoning (TIME-2018)*, volume 120 of *LIPIcs*, pages 8:1–8:16, 2018. `doi:10.4230/LIPIcs.TIME.2018.8`.

2 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 4th Edition*. MIT Press, 2022. URL: `https://mitpress.mit.edu/9780262046305/introduction-to-algorithms`.

3 Rina Dechter, Itay Meiri, and J. Pearl. Temporal Constraint Networks. *Artificial Intelligence*, 49(1-3):61–95, 1991. `doi:10.1016/0004-3702(91)90006-6`.

4 Luke 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)*, pages 155–162, 2009. `doi:10.1109/TIME.2009.25`.

5 Luke Hunsberger. Efficient execution of dynamically controllable simple temporal networks with uncertainty. *Acta Informatica*, 53(2):89–147, 2015. `doi:10.1007/s00236-015-0227-0`.

6 Luke Hunsberger and Roberto Posenato. Speeding up the RUL$^-$ Dynamic-Controllability-Checking Algorithm for Simple Temporal Networks with Uncertainty. In *36th AAAI Conference on Artificial Intelligence (AAAI-22)*, volume 36-9, pages 9776–9785. AAAI Pres, 2022. `doi:10.1609/aaai.v36i9.21213`.

7 Luke Hunsberger and Roberto Posenato. A Faster Algorithm for Converting Simple Temporal Networks with Uncertainty into Dispatchable Form. *Information and Computation*, 293(105063):1–21, 2023. `doi:10.1016/j.ic.2023.105063`.

8 Luke Hunsberger and Roberto 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)*, volume 34, pages 290–300, 2024. `doi:10.1609/icaps.v34i1.31487`.

9 Luke Hunsberger and Roberto Posenato. Foundations of Dispatchability for Simple Temporal Networks with Uncertainty. In *16th International Conference on Agents and Artificial Intelligence (ICAART 2024)*, volume 2, pages 253–263. SCITEPRESS, 2024. `doi:10.5220/0012360000003636`.

10 Paul Morris. A Structural Characterization of Temporal Dynamic Controllability. In *Principles and Practice of Constraint Programming (CP-2006)*, volume 4204, pages 375–389, 2006. doi:10.1007/11889205_28.

11 Paul Morris. Dynamic controllability and dispatchability relationships. In *Int. Conf. on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR-2014)*, volume 8451 of *LNCS*, pages 464–479. Springer, 2014. doi:10.1007/978-3-319-07046-9_33.

12 Paul Morris. The Mathematics of Dispatchability Revisited. In *26th International Conference on Automated Planning and Scheduling (ICAPS-2016)*, pages 244–252, 2016. doi:10.1609/icaps.v26i1.13739.

13 Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *17th Int. Joint Conf. on Artificial Intelligence (IJCAI-2001)*, volume 1, pages 494–499, 2001. URL: https://www.ijcai.org/Proceedings/01/IJCAI-2001-e.pdf.

14 Nicola Muscettola, Paul H. Morris, and Ioannis Tsamardinos. Reformulating temporal plans for efficient execution. In *Proceedings of the Sixth International Conference on Principles of Knowledge Representation and Reasoning*, KR'98, pages 444–452, 1998.

15 Roberto Posenato. STNU Benchmark version 2020, 2020. Last access 2022-12-01. URL: https://profs.scienze.univr.it/~posenato/software/cstnu/benchmarkWrapper.html.

16 Ioannis Tsamardinos, Nicola Muscettola, and Paul Morris. Fast Transformation of Temporal Plans for Efficient Execution. In *15th National Conf. on Artificial Intelligence (AAAI-1998)*, pages 254–261, 1998. URL: https://cdn.aaai.org/AAAI/1998/AAAI98-035.pdf.