A Better Algorithm for Converting an STNU into Minimal Dispatchable Form

Luke Hunsberger
Vassar College, 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. Typically, dispatchability requires inserting conditional *wait* constraints, thereby forming an *Extended* STNU (ESTNU). The number of edges in an ESTNU affects the computational work that must be done during real-time execution. The MinDispESTNU problem is that of finding an equivalent dispatchable ESTNU having a minimal number of edges. Recent work presented an $O(kn^3)$-time algorithm for solving the MinDispESTNU problem, where n is the number of timepoints and k is the number of actions with uncertain durations. A subsequent paper presented a faster $O(n^3)$-time algorithm, but it has been shown to be incomplete. This paper presents a new $O(mn + n^2k + n^2 \log n)$-time algorithm for solving the MinDispESTNU problem, where m is the number of constraints in the network. The correctness of the algorithm is based on a novel theory of the *canonical form of nested diamond structures.* An empirical evaluation demonstrates the order-of-magnitude improvement in performance.

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.2025.12

1 Background

Temporal constraint networks facilitate representing and reasoning about temporal constraints on activities. *Simple Temporal Networks with Uncertainty* (STNUs) 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, as demonstrated by an empirical evaluation.

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. Such edges may be notated as (X, δ, Y) or, if context permits, simply XY. A path from X to Y may be notated by listing its timepoints (e.g., $XUVWY$) or, if the context permits, just XY.

A flexible and efficient *real-time execution* (RTE) algorithm has been defined for STNs that maintains a time window for each timepoint X and, as each X is executed, propagates constraints only *locally,* to X's *neighbors* in the graph [19, 15]. An STN is *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. A consistent STN is dispatchable if and only if each pair of timepoints connected by a path in the 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) [12]. Efficient algorithms for generating equivalent dispatchable STNs having a *minimal number of edges* have been presented [19, 15]. Having fewer edges is important since it lessens real-time computations done during execution.

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 $C - A \in [x, y]$. Each STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ has a corresponding graph $\mathcal{G} = (\mathcal{T}, \mathcal{E}_{\rm o}, \mathcal{E}_{\rm lc}, \mathcal{E}_{\rm uc})$, where $(\mathcal{T}, \mathcal{E}_{\rm o})$ is the graph for the STN $(\mathcal{T}, \mathcal{C})$, and $\mathcal{E}_{\rm lc}$ and $\mathcal{E}_{\rm uc}$ 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 $\mathcal{E}_{\rm lc}$ 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 $\mathcal{E}_{\rm uc}$ 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 dynamic strategy is one whose execution decisions can react to observations of contingent executions, but without advance knowledge of future events. Many polynomial-time DC-checking algorithms have been presented [11, 1, 5], the fastest having a worst-case time-complexity of $O(mn + k^2 n + kn \log n)$, where n, m and k are the numbers of timepoints, ordinary constraints, and contingent links. Many DC-checking algorithms generate a kind of conditional constraint called a *wait* [14, 10, 5]. Although not necessary for DC-checking [1], wait constraints are needed for STNU dispatchability, as follows.

An STNU augmented with a set of waits, $\mathcal{E}_w$, is called an *Extended STNU* (ESTNU) [11]. A real-time execution algorithm for ESTNUs, called RTE*, has been defined that provides maximum flexibility while requiring minimal real-time computation [11, 7]. An ESTNU is dispatchable if every run of the RTE* algorithm is guaranteed to satisfy all of its constraints no matter how the contingent durations turn out. Equivalently, an ESTNU is dispatchable if and only if all of its STN *projections* are dispatchable (as STNs) [11]. (A projection of an ESTNU is the STN that results from fixing the durations of its contingent links.) The fastest algorithm for generating equivalent dispatchable ESTNUs is the $O(mn + kn^2 + n^2 \log n)$-time $\texttt{FD}_{\texttt{STNU}}$ algorithm [6], but it provides no guarantee about the number of edges in its output.

The MinDispESTNU problem: For any given dispatchable ESTNU $\mathcal{G}$, find an equivalent dispatchable ESTNU $\mathcal{G}'$ having a minimal number of edges. The $\texttt{minDisp}_{\texttt{ESTNU}}$ algorithm [7] solves the MinDispESTNU problem in $O(kn^3)$ time. A faster $O(n^3)$-time algorithm, called $\texttt{fastMinDisp}_{\texttt{ESTNU}}$ [8], was later found to be incomplete.

This paper. Section 2 summarizes the $\texttt{minDisp}_{\texttt{ESTNU}}$ and $\texttt{fastMinDisp}_{\texttt{ESTNU}}$ algorithms. Section 3 then presents a new algorithm, `betterMinDispESTNU`, that solves the MinDispESTNU problem in $O(mn + n^2 k + n^2 \log n)$ time. It employs a novel approach to generating so-called *stand-in* edges. The correctness of the algorithm is based on a new theory of the *canonical form of nested diamond structures,* which is detailed in Hunsberger and Posenato [9]. Section 4 presents an empirical evaluation that demonstrates that `betterMinDispESTNU` achieves an order-of-magnitude speedup over $\texttt{minDisp}_{\texttt{ESTNU}}$ in practice.

Figure 1 Stand-in edges entailed by labeled edges associated with contingent links

2 Overview of Existing Algorithms

The $\texttt{minDisp}_{\text{ESTNU}}$ algorithm [7] takes a dispatchable ESTNU $\mathcal{E} = (\mathcal{T}, \mathcal{E}_{\text{o}}, \mathcal{E}_{\text{lc}}, \mathcal{E}_{\text{uc}}, \mathcal{E}_{w})$ as its only input and generates as its output an equivalent dispatchable ESTNU having a minimal number of edges. It has four steps: (1) compute the set of so-called *stand-in* edges (i.e., ordinary edges that are entailed by various combinations of ESTNU edges) and insert them into the graph; (2) apply an STN-dispatchability algorithm to the resulting set of ordinary edges, thereby generating a dispatchable STN subgraph; (3) remove any remaining stand-in edges; and (4) remove any wait edges that are not needed for dispatchability. The $O(kn^3)$ worst-case time complexity of the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm is dominated by Step 1. Therefore, our new, faster algorithm modifies only that step, achieving an order-of-magnitude reduction in the overall worst-case time complexity. The following paragraphs summarize Step 1 of the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm, as implemented by its `genStandIns` helper algorithm.

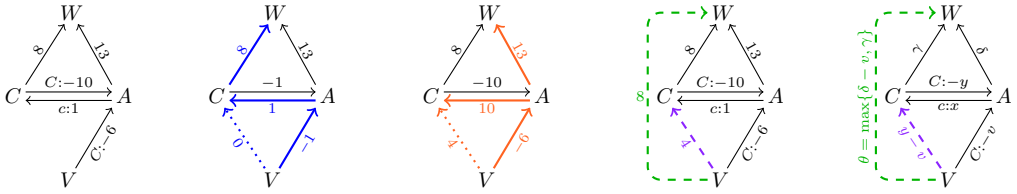
Generating Stand-in Edges

Following Morris [11], an ESTNU is dispatchable if all of its STN projections are dispatchable (as STNs). Equivalently, in each STN projection, each pair of timepoints V and W that are connected by a path must be connected by a *shortest vee-path* (SVP) (i.e., a shortest path comprising zero or more negative edges followed by zero or more non-negative edges) [12]. A key insight behind the $\texttt{minDisp}_{\text{ESTNU}}$ algorithm is that in different projections, the shortest vee-paths from V to W may take different routes, employ different labeled edges, and have different lengths. The *longest* SVP from V to W across all projections determines an *ordinary* constraint, represented by a stand-in edge, that must be satisfied by every valid execution strategy. The $\texttt{minDisp}_{\text{ESTNU}}$ algorithm generates stand-in edges in two phases: (1) those entailed by individual labeled edges; and (2) those entailed by *VACW diamond* structures.

Stand-in edges entailed by individual labeled edges. Each LC, UC or wait edge entails a (weaker) ordinary edge. For example, consider the labeled edges associated with the contingent link $(A, 1, 10, C)$ in Figure 1. The LC edge $(A, c{:}1, 10)$ represents the possibility that the duration $C - A$ might take on its minimum value 1. Its stand-in edge $(A, 10, C)$ represents the (modeled) certainty that $C - A$ will be at most 10. Similarly, the UC edge $(C, C{:}{-}10, A)$ represents the possibility that $C - A$ might take on its maximum value 10, while its stand-in edge $(C, -1, A)$ represents the certainty that $C - A$ will be at least 1. Finally, the wait edge $(V, C{:}{-}6, A)$ represents the conditional constraint that, as long as C remains unexecuted, V must wait until 6 after A. Its stand-in edge $(V, -1, A)$ represents that V must unconditionally wait at least 1 after A, since C cannot execute before then.

More generally, for any contingent link (A, x, y, C), the LC edge $(A, c{:}x, C)$ entails the stand-in edge (A, y, C); the UC edge $(C, C{:}{-}y, A)$ entails the stand-in edge $(C, -x, A)$; and any wait edge $(V, C{:}{-}v, A)$ entails the stand-in edge $(V, -x, A)$, as seen in Figure 1.

Stand-in edges entailed by *VACW* diamond structures. The $\texttt{minDisp}_{\text{ESTNU}}$ algorithm uses its `genStandIns` helper algorithm to compute stand-in edges arising from diamond structures. Figure 2a shows a typical *VACW* diamond, which involves the LC and UC edges associated with a contingent link $(A, 1, 10, C)$, a wait edge $(V, C{:}{-}6, A)$, and some ordinary



(a) ESTNU edges **(b)** If $C - A = 1$ **(c)** If $C - A = 10$ **(d)** Stand-in edges **(e)** General case

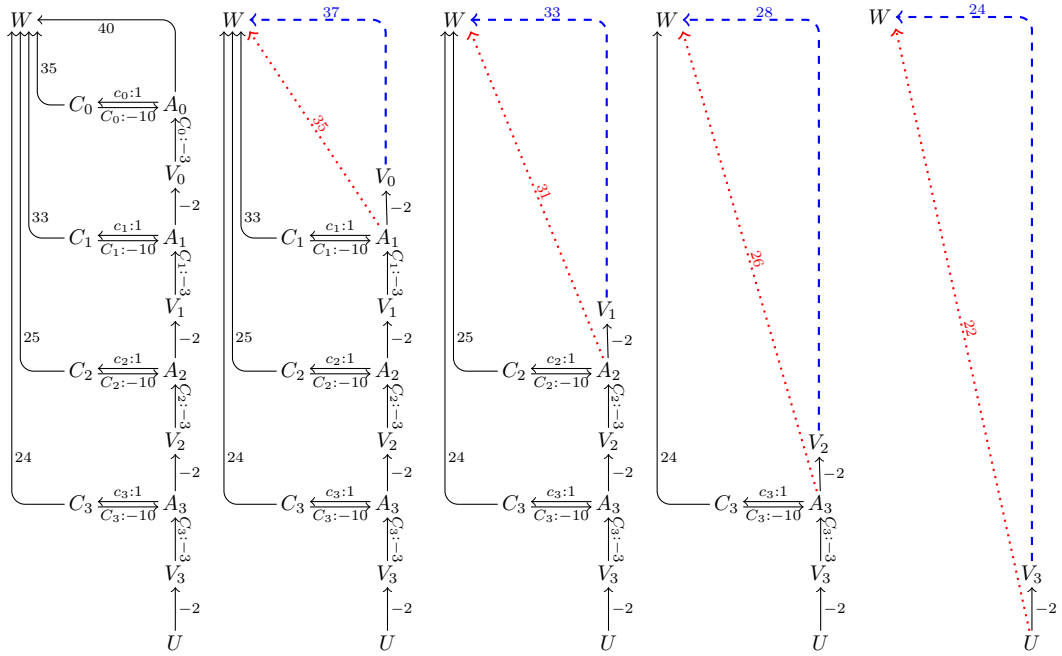
Figure 2 (Dashed) stand-in edges entailed by a *VACW* diamond structure

edges aimed at a timepoint W. Figure 2b shows that in the projection where $C - A = 1$, the shortest path from V to W has length 8, and the shortest path from V to C has length 0. Figure 2c shows that in the projection where $C - A = 10$, the shortest path from V to W has length 7, and the shortest path from V to C has length 4. Figure 2d introduces (dashed) *stand-in* edges to reflect that, across all projections, where $C - A \in [1, 10]$, the shortest path from V to W has length *at most* 8, while the shortest path from V to C has length *at most* 4. These stand-in edges represent ordinary constraints that must be satisfied by any valid dynamic execution strategy. Figure 2e shows the general case where the stand-in edge from V to W has length $\theta = \max\{\delta - v, \gamma\}$, and the stand-in edge from V to C has length $y - v$, the latter being termed an application of the *VAC rule* [9].

Stand-in edges entailed by nested diamonds. The main focus of `genStandIns` is on computing stand-in edges entailed by individual *VACW* diamond structures. But diamond structures can also be *nested.* In particular, in any *VACW* diamond, the subpath from A to W may contain a stand-in edge derived from a nested diamond. However, because the activation timepoints appearing in a nested diamond structure are subject to a strict order (as shown elsewhere [9]), diamonds can only be nested to a maximum depth of k. For this reason, the `genStandIns` algorithm does up to k iterations, each addressing one level of potential nesting. Each iteration of `genStandIns` involves two steps: (1) exploring $O(kn^2)$ individual *VACW* diamonds (k choices for the contingent link, and n choices for both V and W); and then (2) calling Johnson's algorithm [2] to update the APSP distance matrix to accommodate stand-in edges generated by the first step. Figure 3 shows how `genStandIns` deals with a sample quadruply nested diamond structure. The innermost diamond, $V_0\,A_0\,C_0\,W$, is explored during the first iteration, yielding the blue, dashed stand-in edge $(V_0, 37, W)$, shown in Figure 3b, where $\theta_0 = \max\{40 - 3, 35\} = 37$.[1] Johnson's algorithm then updates the APSP distance matrix, setting $d(A_1, W) = 35$, indicated by the red, dotted line in Figure 3b. The next iteration considers $V_1\,A_1\,C_1\,W$, which uses the new subpath from A_1 to W of length 35 to generate the blue, dashed stand-in edge $(V_1, 33, W)$, shown in Figure 3c, where $\theta_1 = \max\{35 - 3, 33\} = 33$. Johnson's algorithm then updates $d(A_2, W)$ to 31, indicated by the red, dotted line in Figure 3c. The third iteration generates the blue, dashed stand-in edge $(V_2, 28, W)$, since $\theta_2 = \max\{31 - 3, 25\} = 28$; and the red, dotted line from A_3 to W indicates the subsequent update $d(A_3, W) = 26$. Finally, as shown in Figure 3d, the last iteration generates the blue, dashed stand-in edge $(V_3, 24, W)$, since $\theta_3 = \max\{26 - 3, 24\} = 24$; while the red, dotted line from U to W indicates the update $d(U, W) = 22$.

The complexity of $\texttt{minDisp}_{\texttt{ESTNU}}$ is driven by the $O(kn^3)$-time complexity of `genStandIns`,

[1] As seen in Figure 1, each labeled edge itself entails a corresponding stand-in edge, not shown in Figure 3. Those stand-in edges ensure that there are ordinary subpaths from each A_i to W, and from each C_i to W, which implies that all of the *VACW* diamonds in Figure 3 would be processed during each iteration of `genStandIns`. However, the stand-in edges shown in Figure 3 are the strongest ones.



(a) Initial **(b)** Iteration 1 **(c)** Iteration 2 **(d)** Iteration 3 **(e)** Iteration 4

Figure 3 How `genStandIns` processes nested diamonds, where stand-in edges derived from individual *VACW* structures are shown in blue, and those computed by Johnson's algorithm in red

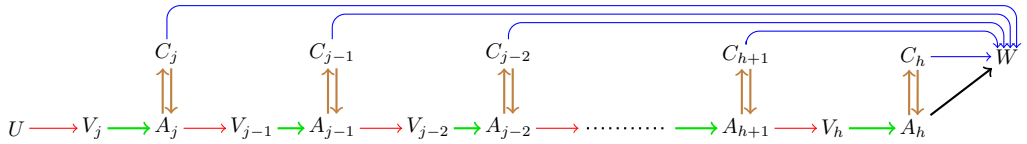


Figure 4 Canonical form of a nested diamond structure $\mathcal{S}_{uw}$ (contingent links in brown, waits in green, negative edges in red, non-negative edges in blue, and an ordinary vee-path in black)

which derives from its up to k calls of Johnson's algorithm on up to $O(n^2)$ edges.

2.1 Canonical Form of Nested Diamond Structures

The authors presented a novel, rigorous theory of the *canonical form of nested diamond structures* [9] that provides a foundation for understanding the dispatchability of ESTNUs and formally proving the correctness of the `minDisp`$_{\text{ESTNU}}$ algorithm. It also highlights features of such structures that suggest new approaches to solving the MinDispESTNU problem.

Central to any such algorithm is computing, for each pair of timepoints U and W, the strongest ordinary constraint entailed by ESTNU paths from U to W, notated as $d_*(U, W)$. For the ESTNU in Figure 3, $d_*(U, W) = 22$ (cf. the red dotted line in Figure 3e). The theory confirms that each value $d_*(U, W)$ that derives from nested diamonds must have an associated structure, notated as $\mathcal{S}_{uw}$, whose form is illustrated in Figure 4. In particular, $\mathcal{S}_{uw}$ comprises a sequence of contingent links, shown in brown, connected by different kinds of paths. From each contingent timepoint C_i, there is a path of non-negative ordinary edges from C_i to W, shown in blue. Between consecutive pairs of activation timepoints A_f and A_g there is a *negOrdWait* path (i.e., a path comprising zero or more negative ordinary edges,

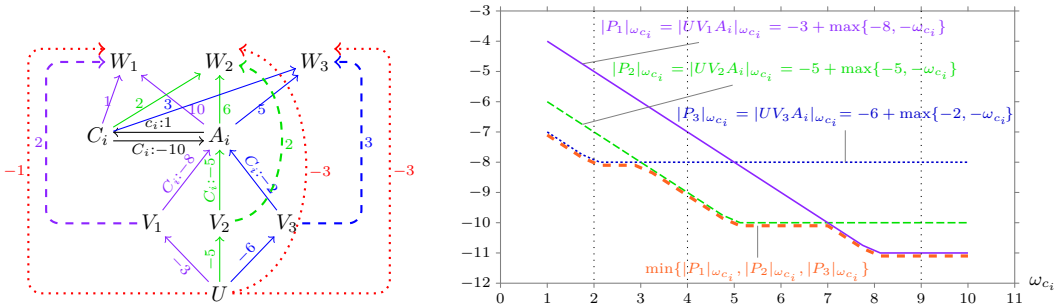


Figure 5 Three *negOrdWait* paths from U to A_i (in purple, green and blue) that determine the values of $d_*(U, W_1)$, $d_*(U, W_2)$ and $d_*(U, W_3)$, indicated by red dotted arrows. Stand-in edges are dashed. Other stand-in edges (e.g., from V_1 to W_2) are not shown.

shown in red, followed by a single wait edge, shown in green). There is also a *negOrdWait* path from U to the leftmost activation timepoint A_j. Finally, the path from the rightmost activation timepoint A_h to W is an ordinary path, shown in black, that is a shortest vee-path (SVP). The path from U to W that passes through all of the activation timepoints is called the *spine* of the structure. For this paper, the following properties are particularly important:

- In the situation/projection where each contingent duration along the spine satisfies $C_i - A_i = \delta_i - \gamma_i = d_*(A_i, W) - d(C_i, W)$, the length of the spine is $d_*(U, W)$.
- The *negOrdWait* paths between consecutive pairs of activation timepoints, across all canonical structures, puts the entire set of activation timepoints into a strict partial order.

2.2 Error in the fastMinDisp$_{\text{ESTNU}}$ Algorithm

Recent work [8] presented an algorithm, called fastMinDisp$_{\text{ESTNU}}$, that aimed to take advantage of certain features of nested diamonds. In particular, it exploited the fact that activation timepoints participating in nested diamonds fall into a strict partial order. That enabled processing them in a single iteration, instead of the k iterations in the minDisp$_{\text{ESTNU}}$ algorithm. Unfortunately, that work made an incorrect assumption. Although it is true that for any given canonical structure it suffices to include only one wait edge terminating at each activation timepoint along the spine, it is **not** the case that *all* of the canonical structures that include some activation timepoint A_i necessarily employ the *same* wait edge terminating at A_i. Instead, as illustrated in Figure 5, different wait edges terminating at A_i may be needed in different canonical structures. In the figure, there are three overlapping canonical structures that each use the contingent link $(A_i, 1, 10, C_i)$: one from U to W_1 (in purple), one from U to W_2 (in green), and one from U to W_3 (in blue). For $d_*(U, W_1)$, the projection where $C_i - A_i = d_*(A_i, W_i) - d(C_i, W_1) = 10 - 1 = 9$ is determinative; and in that projection the shortest path from U to W_1 is through V_1 with length $d_*(U, W_1) = -1$, indicated by the red dotted arrow. The dashed, purple stand-in edge $(V_1, 2, W_1)$ has length 2, since the wait edge $(V_1, C_i{:}-8, A_i)$ has length -8 in that projection. For $d_*(U, W_2)$, the projection where $C_i - A_i = 6 - 2 = 4$ is determinative; and in that projection, the shortest path from U to W_2 is through V_2 with length $d_*(U, W_2) = -3$. The green, dashed stand-in edge $(V_2, 2, W_2)$ has length 2, since the wait edge $(V_2, C_i{:}-5, A_i)$ has length -4 in that projection. For $d_*(U, W_3)$, the projection where $C_i - A_i = 5 - 3 = 2$ is determinative; and in that projection, the shortest path from U to W_3 is through V_3 with length $d_*(U, W_3) = -3$. The blue, dashed stand-in edge $(V_3, 3, W_3)$ has length 3, since the wait edge $(V_3, C_i{:}-2, A_i)$ has length -2 in that projection. The righthand side of Figure 5 plots the lengths of the three paths from U

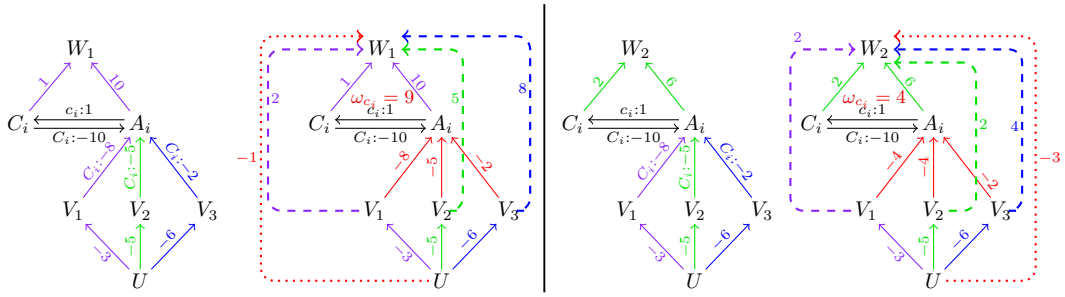


Figure 6 Computing $d_*(U, W_1)$ (left) and $d_*(U, W_2)$ (right) by back-propagation in the OW-graph

to A_i as functions of the contingent duration $\omega_{c_i} = C_i - A_i$. It confirms that for different values of ω_{c_i}, different paths are shortest between U and A_i. As a result, each $d_*(U, W_f)$ value is based on a different path from U to A_i.

In general, for each terminus W_f, the value $d_*(U, W_f)$ is determined by the projection where $C_i - A_i = d_*(A_i, W_f) - d(C_i, W_f)$. Since these durations/projections may be different for different W_f, the wait edges terminating at A_i may provide different shortest vee-paths in different projections. Although this example shows that the `fastMinDisp`$_{\text{ESTNU}}$ algorithm does not necessarily solve the MinDispESTNU problem, it also suggests an alternative way to approach the computation of $d_*(U, W_f)$ values that results in a more efficient (and correct) algorithm for solving the MinDispESTNU problem, which is the subject of the next section.

3 A New Approach to Generating Stand-in Edges

Figure 6 illustrates our new approach to efficiently generating stand-in edges derived from nested diamond structures. It uses the following feature of the canonical form of nested diamonds: in the situation where the duration of each participating contingent link (A_i, x_i, y_i, C_i) is given by $C_i - A_i = \delta_i - \gamma_i = d_*(A_i, W) - d(C_i, W)$, the length of the path from U to W along the spine of the canonical structure equals $d_*(U, W)$. Crucially, these durations are *fixed* for a given W. Therefore, the problem of activation timepoints, A_j and A_i, that are consecutive in multiple overlapping canonical structures employing different wait edges in different structures, can effectively be sidestepped by computing all of the $d_*(U, W)$ values for a fixed W. To do so, our new algorithm *backtracks* from W along shortest paths in the OW-graph (i.e., the graph comprising the ordinary and wait edges from the ESTNU) where wait edges, as they are encountered, are projected using the above-mentioned durations.

On the left of the figure, backtracking from W_1 encounters the activation timepoint A_i, where $d_*(A_i, W_1) = 10$ and $d(C_i, W_1) = 1$, where the determinative duration is $\omega_{c_i} = 10 - 1 = 9$. In this situation, the wait edges terminating at A_i project onto the red edges shown in the middle-left of the figure. In this projection, the path $UV_1A_iW_1$ is shortest, with a length of $-3 - 8 + 10 = -1$, indicated by the red, dotted arrow. The corresponding stand-in edge from V_1 to W_1 is shown as dashed and purple. The dashed stand-in edges emanating from V_2 (green) and V_3 (blue) are also generated, but do not contribute to $d_*(U, W_1)$.

On the righthand side of the figure, backtracking from W_2 encounters A_i and yields the duration $\omega_{c_i} = 6 - 2 = 4$. In this situation, the wait edges project to the red edges shown on the far right. Although each wait edge generates a stand-in edge, the one from V_2 to W_2 provides the shortest path (dotted, red) from U to W_2, which determines $d_*(U, W_2) = -3$.

Pseudocode for our new algorithm for generating stand-in edges entailed by nested diamond structures is given as Algorithm 1. (Appendix A provides pseudocode for all

Algorithm 1 `betterGenStandIns`: Better Algorithm for Generating Stand-in Edges Entailed by Nested Diamonds

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{\rm lc} \cup \mathcal{E}_{\rm uc} \cup \mathcal{E}_w)$, a dispatchable ESTNU graph
Output: $(\mathcal{E}_{si}, d)$, where $\mathcal{E}_{si}$ is a set of stand-in edges; and d is the updated distance matrix

1 $\mathcal{E}_{si} :=$ `getInitStandins`$(\mathcal{G})$ *//Stand-in edges entailed by individual labeled edges and VAC rule*
2 $f :=$ `bellmanFord`$(\mathcal{G}_{ow})$ *//Potential function for OW-graph, $\mathcal{G}_{ow} = (\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{si} \cup \mathcal{E}_w)$*
3 $d :=$ `johnson`$((\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{si}))$ *//Compute the APSP distance matrix for $(\mathcal{T}, \mathcal{E}_{\rm o} \cup \mathcal{E}_{si})$*
4 **foreach** $W \in \mathcal{T}$ **do**
//Init min priority queue, where priority(T) = current estimate of $d_*(T, W)$ re-weighted by f
5 $\mathcal{Q} :=$ new min priority queue
6 $priority := (\infty, \ldots, \infty)$ *//For tracking priority of timepoints in the queue*
7 $\mathcal{Q}.insert(W, 0)$; $priority[W] := 0$
8 $needStandIn2W := \emptyset$ *//$T \in needStandIn2W$ means "need stand-in edge from T to W"*
9 **while** $\neg\mathcal{Q}.empty()$ **do**
10 $(T, \delta^*_{tw}) := \mathcal{Q}.extractMin()$ *//$\delta^*_{tw} = d_*(T, W)$ reweighted by potential function f*
11 $\delta_{tw} := -f(T) + \delta^*_{tw} + f(W)$ *//$\delta_{tw} = d_*(T, W)$ (un-reweighted)*
12 $d(T, W) := \delta_{tw}$ *//Update distance matrix*
13 **foreach** $(R, \delta_{rt}, T) \in (\mathcal{E}_{\rm o} \cup \mathcal{E}_{si})$ **do** *//Back-propagate along ordinary edges*
14 $\delta^*_{rw} := (f(R) + \delta_{rt} - f(T)) + \delta^*_{tw}$ *//Possible new estimate of $d_*(R, W)$ (re-weighted)*
15 **if** $\delta^*_{rw} < priority[R]$ **then** *//New estimate of $d_*(R, W)$ shorter*
16 $needStandIn2W := needStandIn2W \setminus \{R\}$ *//No stand-in edge RW needed*
17 $\mathcal{Q}.insertOrDecreaseKey(R, \delta^*_{rw})$; $priority[R] := \delta^*_{rw}$
18 **if** $T = A$ *is an activation timepoint for a contingent link* (A, x, y, C) **then**
19 $\omega_c := d(A, W) - d(C, W)$ *//ω_c = contingent duration that determines d^* values*
20 **if** $\omega_c \in (x, y]$ **then** *//Condition for generating a non-redundant stand-in edge*
21 **foreach** $(V, C{:}{-}v, A) \in \mathcal{E}_w$ **do** *//Back-propagate along incoming wait edges*
22 $v_{\omega_c} := \max\{-\omega_c, -v\}$ *//Length of wait edge in projection ω_c*
23 $v^*_{\omega_c} := f(V) + v_{\omega_c} - f(A)$ *//Re-weighted length in projection ω_c*
24 $\delta^*_{vw} := v^*_{\omega_c} + priority[A]$ *//δ^*_{vw} = possible new estimate of $d_*(V, W)$*
25 **if** $\delta^*_{vw} \leq priority[V]$ **then**
26 $needStandIn2W := needStandIn2W \cup \{V\}$ *//Need stand-in edge VW*
27 $\mathcal{Q}.insertOrDecreaseKey(V, \delta^*_{vw})$; $priority[V] := \delta^*_{vw}$
28 **foreach** $T \in needStandIn2W$ **do**
29 $\delta_{tw} := -f(T) + priority[T] + f(W)$ *//Actual value of $d_*(T, W)$*
30 $\mathcal{E}_{si} := \mathcal{E}_{si} \cup \{(T, \delta_{tw}, W)\}$ *//Accumulate stand-in edge*
31 **return** $(\mathcal{E}_{si}, d)$

$\texttt{minDisp}_{\texttt{ESTNU}}$ procedures updated to use Algorithm 1.) Algorithm 1 works as follows.

Initialization (Lines 1-3). The `getInitStandins` algorithm (a helper for $\texttt{minDisp}_{\texttt{ESTNU}}$) is called to generate stand-in edges entailed by individual labeled edges (cf. Figure 1) or from applications of the VAC rule (cf. Figure 2e). Next, the Bellman-Ford algorithm [2] is called to compute a solution to the STN, $\mathcal{G}_{ow}$, that comprises the ordinary and wait edges from $\mathcal{G}$, ignoring any alphabetic labels. That solution, f, is then used as a potential function to re-weight the edges in $\mathcal{G}_{ow}$ to have non-negative values, thereby enabling the use of Dijkstra's algorithm [2] to guide the subsequent back-tracking from each W. Finally, Johnson's algorithm [2] is used to compute the initial distance matrix for ordinary paths.

Main **foreach** *Loop (Lines 4-30).* Each iteration of the main **foreach** loop processes a single timepoint W. It uses a modified version of Dijkstra's algorithm to back-propagate

from W through the edges in the $\mathcal{G}_{ow}$ graph, aiming to update the distance function d so that by the end of the iteration, for each timepoint T, $d(T, W) = d_*(T, W)$, and all needed stand-in edges terminating at W have been generated.

Iteration initialization (Lines 5-8). First, a minimum priority queue, $\mathcal{Q}$, is initialized. For each timepoint T in the queue, its priority is the current estimate of $d_*(T, W)$, re-weighted by the potential function f. In particular, the priority of T is given by: $f(T) + \delta_{tw} - f(W)$. Initially, the queue contains only W, with a priority of 0. The n-vector, *priority* enables anytime access to the priorities of timepoints in the queue.

Next, a set *needStandIn2W* is initialized. It is used to keep track of timepoints T for which a stand-in edge from T to W will need to be generated. If the current estimate of $d_*(T, W)$ derives from a path (1) that forms the spine of a canonical diamond structure; and (2) whose first edge is a wait edge, then T is added to *needStandIn2W*, at Line 26. However, should subsequent propagation discover a shortest path from T to W for which no stand-in edge is needed, then T is removed from *needStandIn2W*, at Line 16. Since the status of a given timepoint T may change during the algorithm, stand-in edges are not actually accumulated until the end of the iteration, at Lines 28-30.

Iteration Body (Lines 9–30). The body of each iteration is a **while** loop that carries out the back-propagation from W. At Line 10, a timepoint T is extracted from the queue, along with its priority δ^*_{tw}. At Line 11, the value of $d_*(T, W)$ is extracted from δ^*_{tw} by undoing the re-weighting using the potential function f. (The next section proves the invariant that when a timepoint T is popped from the queue, its priority equals $d_*(T, W)$, re-weighted by the potential function f.) That value is then used to update the distance function d, at Line 12.

Next, Lines 13–17 back-propagate along each incoming ordinary edge (R, δ_{rt}, T). First, at Line 14, a possible new estimate of $d_*(R, W)$ using a path from R to T to W, re-weighted using the potential function f, is computed and stored in δ^*_{rw}. (Note that $f(R) + \delta_{rt} - f(T)$ is the re-weighted length of the incoming edge from R to T.) If that estimate is less than the current priority of R (cf. Line15), then R is removed from *needStandIn2W* to reflect that this newly found shortest path from R to W does not begin with a wait edge and, hence, does not require a stand-in edge (cf. Line 16). At Line 17, R is inserted into the queue and its priority is updated.

Lines 18–27 carry out the back-propagation along any incoming wait edges, which can only happen if $W = A$ is an activation timepoint for a contingent link (A, x, y, C). Line 19 computes the value of the contingent duration $\omega_c = C - A = \delta - \gamma = d_*(A, W) - d(C, W)$ that determines whether any stand-in edges terminating at W can use this contingent link (cf. Figures 2e and 6). Note that the algorithm relies on the fact that $d(A, W) = d_*(A, W)$ at this point. Line 20 checks whether $\omega_c \in (x, y]$, since otherwise, as shown in Claim 10 of Hunsberger and Posenato [9], it is not necessary to back-propagate along any wait edge coming in to A (i.e., ordinary edges suffice). Line 22 computes the projection of the wait edge in the situation where $C - A = \omega_c$. Line 23 re-weights the projected length using the potential function. Line 24 computes the length of the path from V to T to W in the re-weighted graph. If that length less than or equal to the current key of V in the queue (Line 25), then V is added to *needStandIn2W* (at Line 26) to reflect that a stand-in edge should be generated; and the key for V is updated in the priority queue (at Line 27).

Finally, at the end of the iteration, stand-in edges for all of the flagged timepoints are generated at Lines 28-30.

Correctness of the `betterMinDispESTNU` Algorithm

The correctness of `betterMinDispESTNU` relies on the following properties of the canonical form of nested diamond structures that we have rigorously presented elsewhere [9]. (The claims mentioned below are from that work.) First, for each pair of timepoints U and W, there is a canonical form $\mathcal{S}_{uw}$ that determines the value $d_*(U, W)$. Furthermore, $d_*(U, W)$ equals the length of the *spine* of that structure in the situation where each $C_i - A_i = d_*(A_i, W) - d(C_i, W)$. (See the proof of Claim 8.) Second, using the same techniques as in the proof of Claim 7, we get that for a fixed W, there is a *single* situation ω that is simultaneously *maximal* for *all* $d_*(U, W)$ values (i.e., in the projection determined by ω, the length of the spine of each structure $\mathcal{S}_{uw}$ equals $d_*(U, W)$). For each contingent link (A, x, y, C) appearing in *any* canonical structure $\mathcal{S}_{uw}$ from *any* U to the fixed timepoint W, ω specifies the duration, $\omega_c = C - A = d_*(A, W) - d(C, W)$. Therefore, the `betterGenStandIns` algorithm, as it backtracks from W, can be understood as incrementally computing the durations, $\omega_c = C - A$, for each activation timepoint A that it encounters, based on the accumulated values, $d_*(A, W)$ and $d(C, W)$. It then computes the length of each incoming wait edge $(V, C{:}{-}v, A)$ in that projection (i.e., $\max\{-v, -\omega_c\}$), which is its length in the spine of any structure that uses it.

Worst-Case Time Complexity of the `betterMinDispESTNU` Algorithm

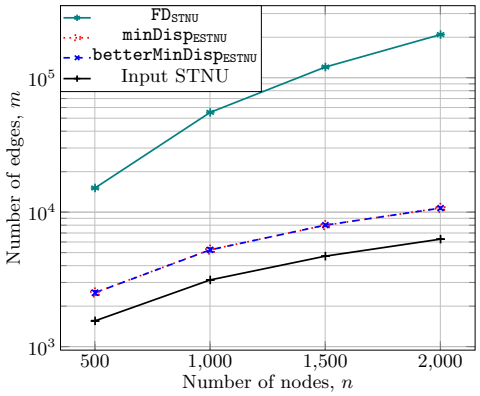
First, let $m = |\mathcal{E}_{\text{o}}|, k = |\mathcal{E}_{\text{lc}}| = |\mathcal{E}_{\text{uc}}|$ and $r = |\mathcal{E}_w| \leq nk$ be the numbers of ordinary, lower-case, upper-case, and wait edges, respectively, in the input ESTNU. Generating stand-in edges for individual labeled edges along with those derived from the VAC rule add $2k{+}2r$ more ordinary edges. Afterward, `betterMinDispESTNU` is applied to the OW-graph which has $m + 2k + 3r$ edges. For each timepoint W, `betterMinDispESTNU` uses a Dijkstra-like back-propagation that runs in $O((m + 2k + 3r + nk) + n \log n)$ time. (At most nk additional stand-in edges can be added during the course of the algorithm.) Therefore, its n iterations can be done in $O((m + 2k + 3r + nk)n + n^2 \log n)$ time, which reduces to $O(mn + n^2k + n^2 \log n)$. For dense graphs, where $m = O(n^2)$, this reduces to $O(n^3)$, but for sparse graphs, for example, where $m = O(n \log n)$ and $k = O(\log n)$, it reduces to $O(n^2 \log n)$.

4 Empirical Evaluations

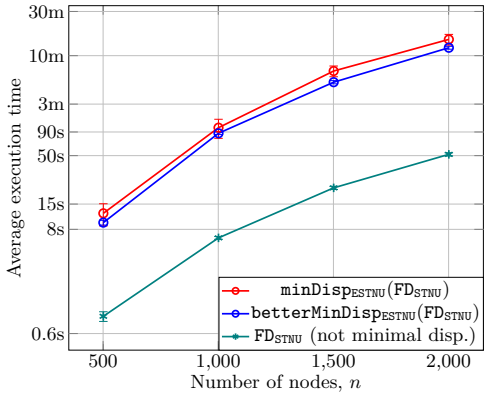
We implemented the `betterMinDispESTNU` algorithm containing the procedure Algorithm 1 in Java—publicly available as part of the CSTNU Tool framework [18]—and evaluated its performance using the STNU benchmark published by Posenato [17]. This benchmark was created using the STNU random generator of the CSTNU Tool framework. The public benchmark comprises 1000 instances, all having the same topology, the *worker-lanes* topology, which simulates the worker lanes of business process modeling [16]. 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, there are extra constraints between nodes in different lanes to represent temporal-coordination constraints among tasks executed by different agents.

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 and, on average, $6.56n - 2.56k - 10$ edges (i.e., $O(n)$ edges). We considered the first 30 instances for each value of n in the benchmark.

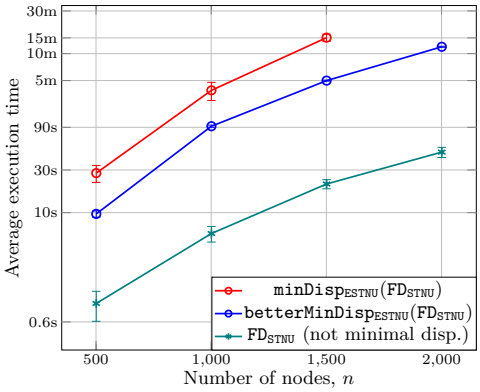
All of the experiments were executed on an OpenJDK JVM 21 configured with 16 GB of



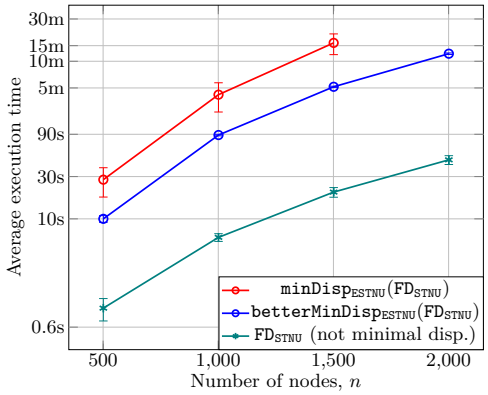
(a) Number of edges in the ESTNUs generated by $\texttt{FD}_{\texttt{STNU}}$, $\texttt{minDisp}_{\texttt{ESTNU}}$ and $\texttt{betterMinDisp}_{\texttt{ESTNU}}$



(b) $\texttt{minDisp}_{\texttt{ESTNU}}$ and $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ performance versus network size



(c) $\texttt{minDisp}_{\texttt{ESTNU}}$ and $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ performance versus network size for instances containing a depth-4 nested diamond structure (cf. Figure 3a)



(d) $\texttt{minDisp}_{\texttt{ESTNU}}$ and $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ performance versus network size for instances containing a depth-6 nested diamond structure

Figure 7 Results of the empirical evaluation of the `betterMinDispESTNU` algorithm

heap memory (parameters `-Xmx16G` and `-Xms16G`), on a Linux computer equipped with two AMD Opteron™ 4334 processors running at 3.1 GHz (6200 BogoMIPS) and 64 GB RAM.

Each DC STNU $\mathcal{G}$ was first pre-processed by the $\texttt{FD}_{\texttt{STNU}}$ dispatchability algorithm to generate an equivalent dispatchable ESTNU, $\mathcal{G}_{\text{fd}}$. Then, the dispatchable ESTNU, $\mathcal{G}_{\text{fd}}$, was fed as input to $\texttt{minDisp}_{\texttt{ESTNU}}$ and $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ to generate equivalent dispatchable ESTNUs having minimal numbers of edges (called μESTNUs) to: (1) confirm that the output μESTNUs were identical; and (2) compare the average execution times.

Surprisingly, during the execution of $\texttt{minDisp}_{\texttt{ESTNU}}$, we observed that *no instances* from the considered benchmarks contain any nested diamond structures. Consequently, there were no opportunities for the $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ algorithm to outperform $\texttt{minDisp}_{\texttt{ESTNU}}$.

Figure 7a shows the average numbers of edges in the input STNUs (black), the dispatchable ESTNUs generated $\texttt{FD}_{\texttt{STNU}}$ (teal), and the minimal dispatchable ESTNUs produced by $\texttt{minDisp}_{\texttt{ESTNU}}$ (dotted red) and $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ (dashed blue). (The dotted red 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 one order of magnitude smaller than in the ESTNUs generated by $\texttt{FD}_{\texttt{STNU}}$. Since the numbers of edges in dispatchable networks directly impact the

performance of real-time execution algorithms, these results demonstrate that $\texttt{minDisp}_{\texttt{ESTNU}}$ and $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ generate dispatchable networks that can be more efficiently executed. We also confirmed that they output the same minimal networks.

Figure 7b plots the computational cost associated with generating μESTNUs. The lower teal line shows the average execution times for $\texttt{FD}_{\texttt{STNU}}$ to generate equivalent dispatchable networks that are typically not μESTNUs. The upper two (red and blue) lines show the average execution times for generating equivalent dispatchable networks having minimal numbers of edges, obtained by applying $\texttt{minDisp}_{\texttt{ESTNU}}$ or $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ to $\mathcal{G}_{\text{fd}}$. As expected, if there are no nested diamond structures, then both algorithms will have essentially equivalent performance since they both end up doing two calls to Johnson's algorithm (or a Johnson-like algorithm).

To assess the impact of nested diamond structures on the performance of the two algorithms, we created two new benchmarks, one comprising random STNU instances that each contain one copy of the depth-4 nested diamond structure depicted in Figure 3a, the other similar to the first, but where the diamond structure has depth 6.

The presence of the depth-4 nested diamond structure in each instance requires the `genStandIns` helper algorithm used by $\texttt{minDisp}_{\texttt{ESTNU}}$ to perform up to five iterations, each taking $O(mn + n^2 \log n)$ time, to generate the appropriate stand-in edges. In contrast, $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ replaces `genStandIns` with Algorithm 1 (`betterGenStandIns`) whose worst-case time complexity is only $O(mn+n^2k+n^2 \log n)$, regardless of how deeply nested the diamond structure may be. We therefore expected to see an especially pronounced difference in average execution times for instances having the depth-6 nested diamond structure.

The results are presented in Figures 7c and 7d. The execution time of $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ ($\texttt{FD}_{\texttt{STNU}}$) (in blue) is significantly less than that of $\texttt{minDisp}_{\texttt{ESTNU}}$ ($\texttt{FD}_{\texttt{STNU}}$) (in red) across all instances. In addition, for instances having 2000 nodes, the execution time of $\texttt{minDisp}_{\texttt{ESTNU}}$ ($\texttt{FD}_{\texttt{STNU}}$) exceeded the 30-minute timeout. Such results confirm that the $\texttt{betterMinDisp}_{\texttt{ESTNU}}$ algorithm is significantly more efficient than the $\texttt{minDisp}_{\texttt{ESTNU}}$ algorithm when the input instances contain nested diamond structures, even when the number of nested diamonds is small. Regarding the depth-6 nested diamond structure, we discovered that, on average, the presence of random constraints among nodes in different lanes and those in the diamond structure sometimes entailed stronger constraints than the stand-in edges associated with the diamond structure and, therefore, the `genStandIns` helper for $\texttt{minDisp}_{\texttt{ESTNU}}$ performs on average five internal iterations, the same as for instances having the quadruply-nested diamond structure.

5 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 required during execution. Therefore, for time-sensitive applications it is important to generate an equivalent dispatchable ESTNU having a minimal number of edges, which we call a μESTNU. This paper modified the only existing algorithm for generating μESTNUs, making it an order-of-magnitude faster. It also showed that a second previously presented algorithm does not in fact solve the MinDispESTNU problem. The new algorithm, `betterMinDispESTNU`, reduced the worst-case time-complexity from $O(kn^3)$ to $O(mn + n^2k + n^2 \log n)$ which, for sparse networks, reduces to $O(n^2 \log n)$.

A Pseudocode

Algorithm 2 `betterMinDispESTNU`: Solving the MinDispESTNU problem

Input: $\mathcal{G} = (\mathcal{T},\ \mathcal{E}_{\text{o}} \cup \mathcal{E}_{\text{lc}} \cup \mathcal{E}_{\text{uc}} \cup \mathcal{E}_{\text{ucg}})$, dispatchable ESTNU
Output: A μESTNU for $\mathcal{G}$

$(\mathcal{E}_o^{\text{si}}, d) :=$ `betterGenStandIns`$(\mathcal{T},\ \mathcal{E}_{\text{o}} \cup \mathcal{E}_{\text{lc}} \cup \mathcal{E}_{\text{uc}} \cup \mathcal{E}_{\text{ucg}})$ *//Compute the set of (ordinary) stand-in edges*
$(\mathcal{T}, \mathcal{E}_{\text{o}}^*, \hat{\mathcal{E}}_{\text{l}}, \hat{\mathcal{E}}_{\text{u}}, \hat{\mathcal{E}}_{\text{ucg}}) :=$ `disp`$_{\text{STN}}(\mathcal{T}, \mathcal{E}_{\text{o}} \cup \mathcal{E}_o^{\text{si}}, \mathcal{E}_{\text{lc}}, \mathcal{E}_{\text{uc}}, \mathcal{E}_{\text{ucg}})$ *//STN dispatchability on ordinary edges, reorienting labeled edges*
$\hat{\mathcal{E}}_{\text{o}}^* := \mathcal{E}_{\text{o}}^* \backslash \mathcal{E}_o^{\text{si}}$ *//Remove any remaining stand-in edges from* $\mathcal{E}_{\text{o}}^*$
$\hat{\mathcal{E}}_{\text{ucg}} := \hat{\mathcal{E}}_{\text{ucg}} \backslash$ `markWaits`$(\mathcal{T}_c, \hat{\mathcal{E}}_{\text{ucg}}, d)$ *//Remove dominated waits*
return $\mathcal{G} = (\mathcal{T}, \hat{\mathcal{E}}_{\text{o}}^* \cup \hat{\mathcal{E}}_{\text{l}} \cup \hat{\mathcal{E}}_{\text{u}} \cup \hat{\mathcal{E}}_{\text{ucg}})$

Algorithm 3 `getInitStandins`: Generate stand-in edges entailed by individual labeled edges

Input: $\mathcal{G} = (\mathcal{T}_x \cup \mathcal{T}_c,\ \mathcal{E}_{\text{o}} \cup \mathcal{E}_{\text{lc}} \cup \mathcal{E}_{\text{uc}} \cup \mathcal{E}_{\text{ucg}})$, 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

$\mathcal{E}_o^{\text{si}} := \emptyset$
foreach $(A, x, y, C) \in \mathcal{L}$ **do** *//Collect stand-in edges for LC, UC and wait edges*
  $\mathcal{E}_o^{\text{si}} := \mathcal{E}_o^{\text{si}} \cup \{(A, y, C), (C, -x, A)\}$ *//Collect stand-in edges for LC and UC edges*
  foreach $(V, C{:}{-}v, A) \in \mathcal{E}_{\text{ucg}}$ **do**
    if $-v \geq -x$ **then** *//Replace weak wait edge by an ordinary edge*
      $\mathcal{E}_{\text{ucg}} := \mathcal{E}_{\text{ucg}} \backslash \{(V, C{:}{-}v, A)\};\ \mathcal{E}_{\text{o}} := \mathcal{E}_{\text{o}} \cup \{(V, -v, A)\}$
    else
      if $-v < -y$ **then** *//Fix misleading wait by adjusting its wait time*
        $\mathcal{E}_{\text{ucg}} := \mathcal{E}_{\text{ucg}} \setminus \{(V, C{:}{-}v, A)\} \cup \{V, C{:}{-}y, A)\}$
      $\mathcal{E}_o^{\text{si}} := \mathcal{E}_o^{\text{si}} \cup \{(V, -x, A), (V, \max\{y - v, 0\}, C)\}$ *//Add stand-in edges for wait edge and from the VAC rule*
return $\mathcal{E}_o^{\text{si}}$

Algorithm 4 `markWaits`: Mark wait edges for removal

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

$\mathcal{E}_{\text{w}}^m := \emptyset$
foreach $(V, C{:}{-}v, A) \in \hat{\mathcal{E}}_{\text{ucg}}$ **do** *//Collect waits dominated by ordinary paths, UC edges, or other waits*
  if $d(V, A) \leq -v$ **or** $d(V, C) < 0$ **then**
    $\mathcal{E}_{\text{w}}^m := \mathcal{E}_{\text{w}}^m \cup \{(V, C{:}{-}v, A)\}$ *//Dominated by an ordinary path or the UC edge*
  else
    foreach $U \in \mathcal{T} \mid \exists (U, C{:}{-}u, A) \in \hat{\mathcal{E}}_{\text{ucg}}$ **do**
      if $d(V, U) < 0$ **and** $d(V, U) - u \leq -v$ **then**
        $\mathcal{E}_{\text{w}}^m := \mathcal{E}_{\text{w}}^m \cup \{(V, C{:}{-}v, A)\}$ *//Dominated by another wait*
return $\mathcal{E}_{\text{w}}^m$

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, 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 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, 2022. doi:10.1609/aaai.v36i9.21213.
6. 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.
7. 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.
8. Luke Hunsberger and Roberto Posenato. Faster Algorithm for Converting an STNU into Minimal Dispatchable Form. In *31st International Symposium on Temporal Representation and Reasoning (TIME 2024)*, volume 318 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:14, 2024. doi:10.4230/LIPIcs.TIME.2024.11.
9. Luke Hunsberger and Roberto Posenato. Canonical Form of Nested Diamond Structures. Technical Report 111/2025, Dipartimento di Informatica - Università degli Studi di Verona, May 2025. URL: https://iris.univr.it/handle/11562/1163111.
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, pages 464–479. 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. Paul H. Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *20th National Conference on Artificial Intelligence (AAAI-2005)*, pages 1193–1198, 2005. URL: https://www.aaai.org/Papers/AAAI/2005/AAAI05-189.pdf.
15. 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, page 444–452, 1998.
16. Object Management Group (OMG). Business process definition metamodel (bpdm), Beta 1. http://www.omg.org, 2007.

469 **17** Roberto Posenato. STNU Benchmark version 2020, 2020. https://profs.scienze.univr.
470 it/~posenato/software/cstnu/benchmarkWrapper.
471 **18** Roberto Posenato. CSTNU Tool: A Java library for checking temporal networks. *SoftwareX*,
472 17:100905, 2022. doi:10.1016/j.softx.2021.100905.
473 **19** Ioannis Tsamardinos, Nicola Muscettola, and Paul Morris. Fast Transformation of Temporal
474 Plans for Efficient Execution. In *15th National Conf. on Artificial Intelligence (AAAI-1998)*,
475 pages 254–261, 1998. URL: https://cdn.aaai.org/AAAI/1998/AAAI98-035.pdf.