Speeding Up the RUL$^-$ Dynamic-Controllability-Checking Algorithm for Simple Temporal Networks with Uncertainty

Luke Hunsberger,[1] **Roberto Posenato**[2]

[1]Department of Computer Science, Vassar College, Poughkeepsie, NY USA,
[2]Dipartimento di Informatica, Università di Verona, Verona, Italy
hunsberger@vassar.edu, roberto.posenato@univr.it

Abstract

A Simple Temporal Network with Uncertainty (STNU) includes real-valued variables, called time-points; binary difference constraints on those time-points; and contingent links that represent actions with uncertain durations. STNUs have been used for robot control, web-service composition, and business processes. The most important property of an STNU is called dynamic controllability (DC); and algorithms for checking this property are called DC-checking algorithms. The DC-checking algorithm for STNUs with the best worst-case time-complexity is the RUL$^-$ algorithm due to Cairo, Hunsberger and Rizzi. Its complexity is $O(mn + k^2n + kn \log n)$, where n is the number of time-points, m is the number of constraints, and k is the number of contingent links. It is expected that this worst-case complexity cannot be improved upon. However, this paper provides a new algorithm, called RUL2021, that improves its performance in practice by an order of magnitude, as demonstrated by a thorough empirical evaluation.

Background

Simple Temporal Networks (STNs) were introduced by Dechter, Meiri, and Pearl (1991) to facilitate reasoning about time. An STN is a pair $(\mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ is a set of real-valued variables called *time-points,* and $\mathcal{C}$ is a set of binary difference constraints on those time-points. Each constraint in an STN has the form, $Y - X \leq \delta$, where $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$. Typically, the number of time-points is denoted by $n = |\mathcal{T}|$, and the number of constraints by $m = |\mathcal{C}|$. Each STN has a corresponding graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, where the time-points serve as nodes, and the constraints correspond to edges. Specifically, for each constraint $(Y - X \leq \delta)$ in $\mathcal{C}$, there is a labeled directed edge, called an *ordinary* edge, $X \xrightarrow{\delta} Y$ in $\mathcal{E}$.

An STN is consistent (i.e., has a solution as a constraint satisfaction problem) if and only if its graph has no negative cycles (Dechter, Meiri, and Pearl 1991). The consistency of an STN can be determined, for example, by the Bellman-Ford (BF) Single-Source Shortest-Paths (SSSP) algorithm (Cormen et al. 2009). This paper uses the version of BF in which a new node $S \notin \mathcal{T}$ is used as the source node. For consistent networks, BF generates a distance function d, where for each time-point $X \in \mathcal{T}$, $d(X)$ equals the length of the shortest path from S to X. (Initially, for each $X \in \mathcal{T}$, $d(X) = 0$, simulating an edge $S \xrightarrow{0} X$.) Such a d will be a solution to the original STN. Otherwise, BF detects a negative cycle.

The algorithms in this paper use variants of Dijkstra's SSSP algorithm (Cormen et al. 2009), propagating either forward from a source node or backward from a sink node. Although Dijkstra typically applies only to STNs whose edge-weights are all non-negative, it may also be used for STNs having some negative edges—if those negative edges either all emanate from or all terminate in a single time-point (Morris 2014). As in Johnson's algorithm (Cormen et al. 2009), a *potential function* can be used to convert edge-weights in a consistent STN to non-negative values, thereby enabling the use of Dijkstra to guide the traversal of shortest paths.

Simple Temporal Networks with Uncertainty

A Simple Temporal Network with Uncertainty (STNU) is an STN augmented with *contingent links* that can be used to represent actions with uncertain durations (Morris, Muscettola, and Vidal 2001). STNUs are used for robot control (Karpas et al. 2015), web-service composition (Franceschetti and Eder 2019), and business processes (Franceschetti and Eder 2020a,b). Faster algorithms for managing STNUs will make them practical for a wider range of applications.

An STNU is a triple $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ where $(\mathcal{T}, \mathcal{C})$ is an STN and $\mathcal{L} = \{(A_i, x_i, y_i, C_i)\}_{0 \leq i < k}$ is a set of *contingent links,* where for each i, $0 < x_i < y_i < \infty$; $A_i, C_i \in \mathcal{T}$; and $C_i \equiv C_j$ iff $i = j$.[1] The number of contingent links (CLs) is denoted by k. For each $(A_i, x_i, y_i, C_i) \in \mathcal{L}$, A_i is its *activation* time-point (ATP), C_i its *contingent* time-point (CTP), and $\Delta_i = y_i - x_i$ the uncertainty in its duration. We also let $\mathcal{T}_C = \{C_i \mid \exists (A_i, x_i, y_i, C_i) \in \mathcal{L}\}$ denote the set of CTPs; and $\mathcal{T}_X = \mathcal{T} \backslash \mathcal{T}_C$ the set of *executable* time-points. A system using an STNU controls the *execution* of the executable time-points (i.e., assigns values to them), but only *observes* the execution of contingent time-points as they occur.

Each STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ has a graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, where the time-points in $\mathcal{T}$ serve as nodes, and the constraints in $\mathcal{C}$ and contingent links in $\mathcal{L}$ together correspond to edges. Specifically, $\mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u$ where:

- for each ordinary constraint $(Y - X \leq \delta)$ in $\mathcal{C}$, there is an *ordinary* edge, $X \xrightarrow{\delta} Y$ in $\mathcal{E}_o$ (as in an STN); and

[1]The notation $X \equiv Y$ represents that X and Y are the same variable, not that their values are equal.

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

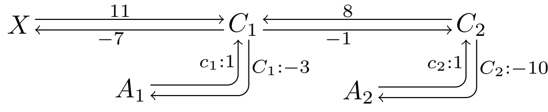


Figure 1: A sample STNU graph

- for each contingent link (A_i, x_i, y_i, C_i) in $\mathcal{L}$, there is a *lower-case* (LC) edge $A_i \xrightarrow{c_i:x_i} C_i$ in $\mathcal{E}_\ell$, and an *upper-case* (UC) edge $C_i \xrightarrow{C_i:-y_i} A_i$ in $\mathcal{E}_u$.

Figure 1 shows the graph for an STNU whose time-points are A_1, C_1, A_2, C_2 and X. The ordinary edges between X and C_1, and between C_1 and C_2 correspond to ordinary constraints (as in an STN). A_1 and A_2 are ATPs, C_1 and C_2 are CTPs, and X, A_1 and A_2 are executable. The LC edges from A_1 to C_1, and A_2 to C_2 represent the uncontrollable possibilities that the durations of the contingent links, $(A_1, 1, 3, C_1)$ and $(A_2, 1, 10, C_2)$, may each turn out to be as low as 1. The UC-edges represent the uncontrollable possibilities that their durations may be as high as 3 and 10, respectively.

For convenience, an ordinary edge $X \xrightarrow{\delta} Y$ may be notated as (X, δ, Y); an LC-edge $A_i \xrightarrow{c_i:x_i} C_i$ as $(A_i, c_i{:}x_i, C_i)$; and a UC-edge $C_i \xrightarrow{C_i:-y_i} A_i$ as $(C_i, C_i{:}{-y_i}, A_i)$; and a *path* may be notated by combining its constituent edges within a single expression. For example, the path containing the edges $(A, c{:}1, C)$, $(C, 5, X)$ and $(X, -7, Y)$ may be notated as $(A, c{:}1, C, 5, X, -7, Y)$. We also adopt the following notation from Hunsberger (2015): The *LO-edges* comprise the LC and ordinary edges (i.e., $\mathcal{E}_\ell \cup \mathcal{E}_o$); the *LO-graph* $\mathcal{G}_{\ell o} = (\mathcal{T}, \mathcal{E}_\ell \cup \mathcal{E}_o)$ contains just the LO-edges; the *OU-edges* comprise the ordinary and UC-edges (i.e., $\mathcal{E}_o \cup \mathcal{E}_u$); and the *OU-graph* $\mathcal{G}_{ou} = (\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_u)$ contains just the OU-edges.

Dynamic Controllability of an STNU

The most important property of an STNU is called *dynamic controllability* (DC) (Morris, Muscettola, and Vidal 2001). An STNU $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ is DC if there exists a dynamic strategy for executing its *executable* time-points that guarantees that all ordinary constraints will be satisfied no matter how the durations of the contingent links turn out. Crucially, a dynamic strategy can *react* to observations of executions of contingent time-points, but its execution decisions cannot depend on advance knowledge of future events. This paper assumes the version of DC in which dynamic strategies can react *instantaneously* to observations of contingent executions (Morris 2006). Thus, execution decisions may depend on past *or present* observations. The formal definition of the DC property appears in Hunsberger and Posenato (2021).

The sample STNU is not DC; (i.e., there is no strategy for executing X, A_1 and A_2 that can guarantee that all ordinary constraints will be satisfied no matter how the contingent durations, $C_1 - A_1$ and $C_2 - A_2$, turn out). However, weakening the edge $(X, 11, C_1)$ to $(X, 14, C_1)$ would make it DC. In fact, the following dynamic strategy would ensure that all ordinary constraints would be satisfied: *Execute X and A_2 at time 0. If C_2 happens to execute before time 6, then execute A_1 at time 6; else, execute A_1 at time $C_2 + 1$.*

Rule	Graphical Representation	Conditions
(NC)	$X \xrightarrow{v} Y \xrightarrow{w} W$, generates $X \overset{v+w}{\dashrightarrow} W$	(none)
(UC)	$X \xrightarrow{v} Y \xrightarrow{C:w} A$, generates $X \overset{C:v+w}{\dashrightarrow} A$	(none)
(LC)	$A \xrightarrow{c:x} C \xrightarrow{w} X$, generates $A \overset{x+w}{\dashrightarrow} X$	$w < 0$
(Cr)	$A \xrightarrow{c:x} C \xrightarrow{K:w} B$, generates $A \overset{K:x+w}{\dashrightarrow} B$	$K \not\equiv C, w < 0$
(LR)	$X \xrightarrow{C:w} A \xrightarrow{c:x} C$, generates $X \overset{w}{\dashrightarrow} A$	$w \geq -x$

Table 1: Edge-generation rules (Morris and Muscettola 2005)

Rule	Graphical representation	Applicability Conditions
R	$P \xrightarrow{v} Q \xrightarrow{w} C_i$, generates $P \overset{v+w}{\dashrightarrow} C_i$	$Q \in \mathcal{T}_X, w < \Delta_i, C_i \in \mathcal{T}_C$
L	$A_j \xrightarrow{c_j:x_j} C_j \xrightarrow{w} C_i$, generates $A_j \overset{x_j+w}{\dashrightarrow} C_i$	$C_j \not\equiv C_i, w < \Delta_i, C_i \in \mathcal{T}_C$
U	$P \xrightarrow{v} C_i \xrightarrow{C_i:-y_i} A_i$, generates $P \overset{\max\{v-y_i,-x_i\}}{\dashrightarrow} A_i$	$(A_i, x_i, y_i, C_i) \in \mathcal{L}$

Table 2: The edge-generation rules for the RUL^- algorithm

Morris (2006) introduced an $O(n^4)$-time DC-checking algorithm based on the edge-generation rules shown in Table 1. (For each rule, the edge generated by the rule is shown as a dashed edge.) That algorithm uses the rules to generate OU-edges that effectively bypass (or "reduce away") LC edges. Once the LC edges have been bypassed, the DC-checking problem reduces to checking whether the OU-graph (augmented with the bypass edges) has a negative cycle. Intuitively, the LC edges are important for that algorithm only insofar as they are able to contribute new edges to the OU-graph. Later, Morris (2014) introduced an $O(n^3)$-time DC-checking algorithm that uses the same rules, but in a different way, back-propagating from *negative nodes* (i.e., nodes having one or more incoming negative edges), aiming to bypass all *negative* OU-edges with non-negative edges. For this algorithm, the input STNU is DC if and only if all negative OU-edges can be reduced away.

The RUL^- DC-Checking Algorithm

Cairo, Hunsberger, and Rizzi (2018) introduced an $O(mn + k^2n + kn\log n)$-time DC-checking algorithm, called RUL^-, that is the fastest so far.[2] Although its complexity for dense grahps is $O(n^3)$, for sparse graphs, whre $k = O(\sqrt{n})$ and $m = O(n \log n)$, its complexity reduces to $O(n^2 \log n)$.

The RUL^- algorithm uses the rules in Table 2. (R, U and L are abbreviations for Relax, Upper and Lower.) These rules differ from those in Table 1 as follows: (1) the RUL^- rules only generate *ordinary* edges; (2) the R rule is the same as the *NC* rule, but with stricter applicability conditions; (3) the L rule is like the *LC* rule, but with *different* conditions; (4) the U rule is similar to the *UC* rule, but is *not* length preserving when $v - y < -x$; and (5) since the RUL^- rules never

[2] More recently, Bhargava and Williams (2019a,b) presented a sound-but-incomplete algorithm that combines features of the Morris 2014 and RUL^- algorithms.

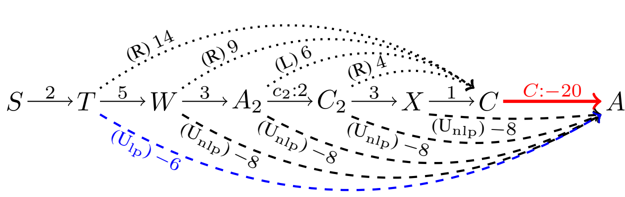


Figure 2: RUL^- generating bypass edges for a UC-edge

generate UC-edges, the *LR* and *Cr* rules are not needed.

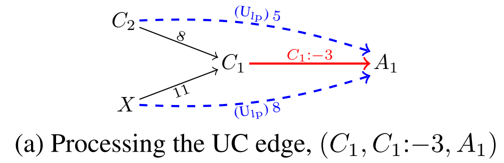
The RUL^- algorithm focuses on the k UC edges, seeking to use the rules from Table 2 to generate (ordinary) edges that bypass the UC edges. Once all UC edges have been bypassed, the DC-checking problem reduces to checking the consistency of the LO-graph (including the bypass edges).

Figure 2 illustrates the two phases of processing for a sample UC edge $(C, C{:}{-20}, A)$, shown in red, associated with a contingent link $(A, 8, 20, C)$. In the first phase, the algorithm propagates backward from C, along edges in the LO-graph, looking for opportunities to apply the R and L rules to generate new edges terminating at C, shown as dotted edges at the top of Figure 2. (The rule used to generate each edge is given in parentheses.) For example, applying the R rule to the edges $(C_2, 3, X)$ and $(X, 1, C)$ generates the (dotted) edge $(C_2, 4, C)$; and then applying the L rule to $(A_2, c_2{:}2, C_2)$ and $(C_2, 4, C)$ generates the (dotted) edge $(A_2, 6, C)$. Since the R and L rules only apply when the righthand edge has length less than $\Delta_C = 20 - 8 = 12$, this backward propagation stops when the edge $T \xrightarrow{14} C$ is generated, since $14 \geq 12$.

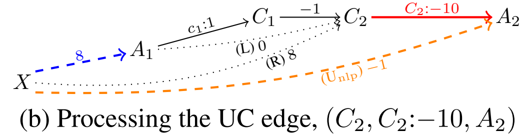
To make the first phase's exploration of paths in the LO-graph efficient, the algorithm uses a potential function, as in Johnson's algorithm, to re-weight the LO-edges to be non-negative. This enables the use of Dijkstra's algorithm to guide the exploration of shortest LO-paths. (More on this later.)

In the second phase, for each (new or pre-existing) LO-edge terminating at C, the algorithm uses the U rule to combine that edge with the (red) UC edge to generate a bypass edge. For example, applying the U rule to $(X, 1, C)$ and $(C, C{:}{-20}, A)$ generates the bypass edge $(X, -8, A)$, shown as a dashed edge at the bottom of Figure 2. This is an example of the *non-length-preserving* case of the U rule, which we call $\mathrm{U_{nlp}}$. In general, this case arises whenever $v - y_i < -x_i$ (equiv., $v < \Delta_i$). In the example, $v - y = 1 - 20 < -8 = -x$ (equiv., $v = 1 < 12 = \Delta_C$). Note that every edge terminating at C in Figure 2, except the last one, has length less than $\Delta_C = 12$. (That is why back-propagation in the first phase stopped at T.) Therefore, every bypass edge, except the one from T to A, has length $-x = -8$. In contrast, because the length of the edge $(T, 14, C)$ is $14 \geq 12 = \Delta_C$, the *length-preserving* case of the U rule, which we call $\mathrm{U_{lp}}$, generates the edge $(T, -6, A)$, where $-6 = 14 - 20$.

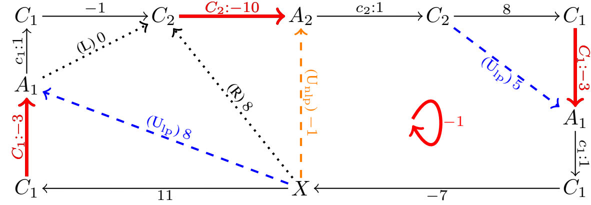
The RUL^- algorithm can now be summarized as follows: (1) it runs Bellman-Ford once to compute an initial potential function for the LO-graph; (2) it propagates backward from each of the k UC-edges along edges in the LO-graph, seeking to generate edges that bypass those UC-edges; (3) after each processing of a UC-edge, it incrementally updates the potential function; (4) if the back-propagation from a UC-edge ever encounters another UC edge that has not already been processed, then it interrupts its processing of the first UC-edge; and (5) after an interruption, which typically generates new edges, the potential function is updated and the processing of the interrupted UC-edge is re-started from scratch.



(a) Processing the UC edge, $(C_1, C_1{:}{-3}, A_1)$



(b) Processing the UC edge, $(C_2, C_2{:}{-10}, A_2)$



(c) The resulting negative cycle in the LO-graph

Figure 3: RUL^- processing the STNU from Figure 1

Figure 3 illustrates how the RUL^- algorithm discovers that the sample STNU from Figure 1 is not DC. In the figure, the UC edges are shown in red. First, back-propagation from the UC edge $(C_1, C_1{:}{-3}, A_1)$ generates the bypass edges $(C_2, 5, A_1)$ and $(X, 8, A_1)$, shown as blue dashed edges in Figure 3a. Next, back-propagation from the UC edge $(C_2, C_2{:}{-10}, A_2)$ generates the bypass edge $(X, -1, A_2)$, shown as an orange dashed edge in Figure 3b. With these bypass edges, there is now a negative cycle in the LO-graph: $(X, -1, A_2, c_2{:}1, C_2, 5, A_1, c_1{:}1, C_1, -7, X)$, as shown in Figure 3c. Hence, the next attempt to update the potential function will fail, signaling that the STNU must be non-DC.

A New Approach to the RUL^- Algorithm

This section introduces a new DC-checking algorithm for STNUs that we call RUL2021. It achieves an order of magnitude improvement in performance over the RUL^- algorithm. Like the RUL^- algorithm, the RUL2021 algorithm aims to generate edges that bypass UC edges; and its back-propagation in the LO-graph is guided by Dijkstra's algorithm, using a potential function created by an initial call to Bellman-Ford, then incrementally updated as new edges are added. However, its substantial differences include:

- It *dramatically reduces* the number of edges that are inserted into the STNU graph, thereby significantly reducing the amount of constraint propagation required by the many calls to Dijkstra. In particular, the new algorithm:
  - only inserts (dashed) edges generated by the $\mathrm{U_{lp}}$ rule.

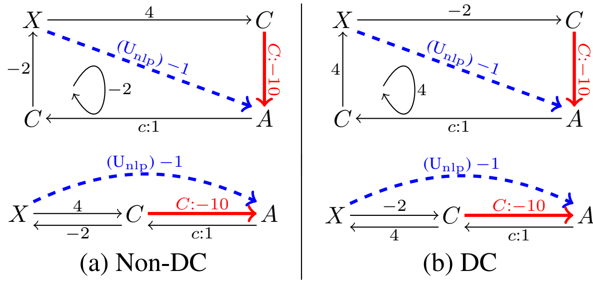


Figure 4: Two scenarios in which back-propagation from the UC edge $(C, C{:}{-10}, A)$ encounters a non-negative cycle $\mathcal{L}$ from C back to C such that $0 \leq |\mathcal{L}| = 2 < 9 = \Delta_C$

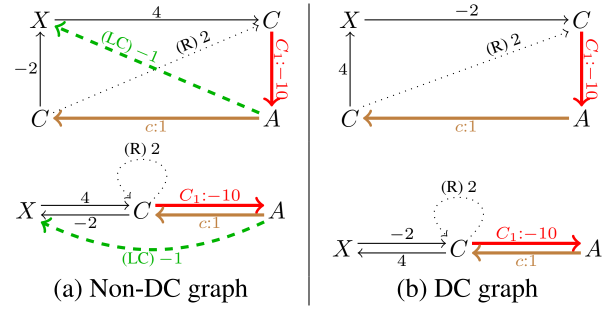


Figure 5: How the new RUL algorithm processes the scenarios from Figure 4 without using the $\mathrm{U_{nlp}}$ rule

- only *accumulates, but does not insert* any (dotted) edges generated by the R or L rules; and
- does not use the (non-length-preserving) $\mathrm{U_{nlp}}$ rule.

As will be seen, *not* using the $\mathrm{U_{nlp}}$ rule sometimes requires making extra calls to Dijkstra to check whether certain *non-negative* cycles in the LO-graph lead to *negative* cycles in the OU-graph, making the STNU not DC. However, our extensive empirical evaluation shows that this cost is more than offset by the overwhelming benefit of reducing the number of edges inserted into the graph.

- The new algorithm keeps track of the work done so far while processing a UC edge so that when any interruptions (i.e., processings of other encountered UC edges) are finished, it can resume processing where it left off, even if the potential function has been updated multiple times in the interim. This enables the new algorithm to be implemented *recursively,* like Morris' 2014 algorithm, making at most k recursive calls to process UC edges, instead of at most $2k$ iterative calls in the RUL^- algorithm.
- When processing a UC edge, the new algorithm does not insert any new edges into the STNU graph until all recursive processing of any interrupting UC edges is completed, thereby requiring fewer updates of the potential function.

The novel features are described in more detail below.

Avoiding the $\mathbf{U_{nlp}}$ rule. The correctness proof for the RUL^- algorithm uses the $\mathrm{U_{nlp}}$ rule for only two purposes: (1) to prove that a cycle of interrupted processings of UC edges implies that the original STNU is not DC; and (2) to deal with the case where back-propagation from a UC edge $(C, C{:}{-y}, A)$ encounters a *non-negative* cycle $\mathcal{L}$ in the LO-graph from C back to C, where $0 \leq |\mathcal{L}| < \Delta_C$. Regarding Case (1), it is true that inserting the edges generated by the $\mathrm{U_{nlp}}$ rule ensures that a cycle of interruptions will be detected the next time the potential function for the LO-graph is updated, but it is not the only way. Instead, as in Morris' 2014 algorithm, it suffices to merely monitor for the presence of a cycle of recursive interruptions and, if such a cycle is ever found, immediately conclude that the network is not DC.

Case (2) is illustrated by the contrasting scenarios shown in Figure 4: one non-DC, one DC. For each scenario, the bottom picture shows the STNU graph and the top shows the cycle being considered. In each scenario, processing the UC edge $(C, C{:}{-10}, A)$ involves back-propagating along a cycle $\mathcal{L}$ from C to X to C, where $0 \leq |\mathcal{L}| = 2 < 9 = \Delta_C$; however, on the left, $\mathcal{L} = (C, -2, X, 4, C)$, whereas on the right, $\mathcal{L} = (C, 4, X, -2, C)$. This slight difference makes the lefthand graph non-DC, and the righthand graph DC.

When the RUL^- algorithm processes the lefthand graph, the UC edge is immediately reduced away by applying the $\mathrm{U_{nlp}}$ rule to the edges $(X, 4, C)$ and $(C, C{:}{-10}, A)$, generating the (dashed) bypass edge $(X, -1, A)$. This creates a negative cycle $(A, c{:}1, C, -2, X, -1, A)$ in the LO-graph, which the algorithm detects when it tries to update the potential function. The key features are that the lengths of the LC edge $(A, c{:}1, C)$ and the generated edge $(X, -1, A)$ sum to zero, while the edge from C to X has negative length. In contrast, although back-propagating from C in the righthand graph also generates the dashed edge $(X, -1, A)$, no negative cycle in the LO-graph arises because, in this scenario, the weight on the edge $(C, 4, X)$ is non-negative. In particular, the cycle $(A, c{:}1, C, 4, X, -1, A)$ has length $4 \geq 0$.

While the RUL^- algorithm uses the $\mathrm{U_{nlp}}$ rule to distinguish the scenarios in Figure 4, our new algorithm distinguishes them without using $\mathrm{U_{nlp}}$. First, as it back-propagates from a UC edge $(C, C{:}{-y}, A)$, the new algorithm keeps track of whether it ever encountered a cycle from C back to C of length less than Δ_C, which we call a *CC loop*. If so, after completing its back-propagation, it then carries out a separate *forward* propagation from C in the LO-graph, looking for opportunities to generate bypass edges for the *lower-case* edge $(A, c{:}x, C)$. This forward propagation is similar to that used by the Morris 2006 algorithm, except that: (1) it only visits LO-edges, not OU-edges; and (2) it only visits nodes that were encountered during the back-propagation from C along paths of length less than Δ_C (i.e., those nodes for which the $\mathrm{U_{nlp}}$ rule would apply if it were being used). If it is able to generate a bypass edge for the LC edge, then the algorithm immediately halts, reporting that the STNU is not DC; otherwise, it resumes normal processing.

Figure 5 illustrates how our new algorithm deals with the scenarios in Figure 4. In each case, back-propagation from C does *not* use the $\mathrm{U_{nlp}}$ rule to generate a (dashed) bypass edge from X to A. Instead, propagation from C continues past X, using the R rule to generate (but not insert) the (dotted) edge/loop $(C, 2, C)$. Since $0 \leq 2 < 9 = \Delta_C$, this CC loop triggers, in each scenario, a separate forward propagation from C, looking for opportunities to bypass the LC edge

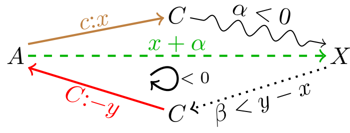
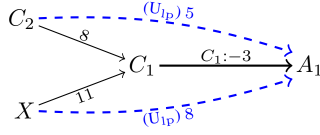
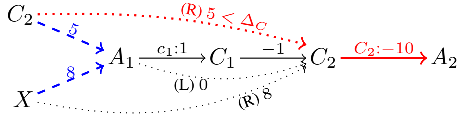


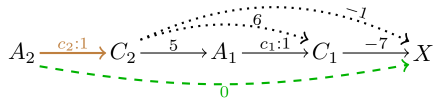
Figure 6: Forward propagation generating an OU-loop of length $x + \alpha + \beta - y < x + 0 + (y - x) - y = 0$



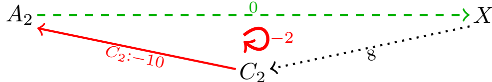
(a) Processing $(C_1, C_1{:}{-3}, A_1)$



(b) Back-propagating from UC edge $(C_2, C_2{:}{-10}, A_2)$ reveals CC loop $(C_2, 5, A_1, c_1{:}1, C_1, -1, C_2)$ of length $5 < \Delta_{C_2}$



(c) Forward propagation from C_2 along LO-edges generates a (green, dashed) bypass edge for the LC-edge $(A_2, c_2{:}1, C_2)$

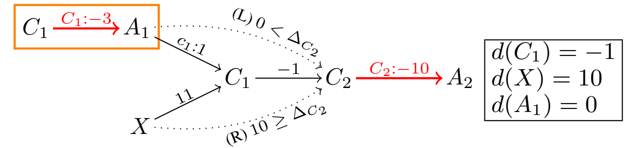


(d) Leading to a negative loop in the OU-graph

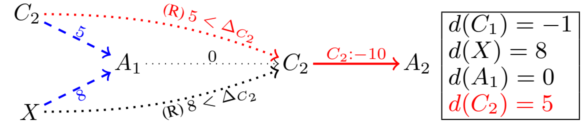
Figure 7: The RUL2021 algorithm's processing of a tighter version of the STNU from Figure 1

$(A, c{:}1, C)$. On the left, forward propagation generates the bypass edge $(A, -1, X)$ using the LC rule from Table 1, which creates a negative loop in the OU-graph, implying that the STNU is not DC. More generally, as shown in Figure 6, any such bypass edge $(A, x+\alpha, X)$ *necessarily* combines with the dotted edge (X, β, C) and the UC edge $(C, C{:}{-y}, A)$ to form a negative loop in the OU-graph, since α must be negative, and $\beta < \Delta_C = y - x$. Our algorithm returns $\perp$ as soon as a bypass edge is found. In contrast, on the right of Figure 5, forward propagation from C in the LO-graph ends at the other occurrence of C, without generating a bypass edge. Hence, our algorithm would resume normal processing—if there were other UC edges to process.

Next, recall Figure 3 which shows how the RUL^- algorithm processes the STNU from Figure 1. Recall further that RUL^- uses the $\mathrm{U_{nlp}}$ rule to generate the edge $(X, -1, A_2)$ which contributes to a negative cycle in the LO-graph. Figure 7 shows how our new algorithm would process this STNU without using the $\mathrm{U_{nlp}}$ rule. First, the UC-edge $(C_1, C_1{:}{-3}, A_1)$ is processed as shown in Figure 7a. Next, as shown in Figure 7b, back-propagation from C_2 detects a (red, dotted) loop from C_2 back to C_2 of length $5 < \Delta_{C_2}$. This



(a) Processing $(C_2, C_2{:}{-10}, A_2)$ first encounters interruption at A_1



(b) Resuming after processing $(C_1, C_1{:}{-3}, A_1)$ (as in Figure 7a)

Figure 8: The new algorithm dealing with an interruption

triggers a separate forward propagation from C_2 which, as shown in Figure 7c, generates a bypass edge (dashed, green) for the LC-edge $(A_2, c_2{:}1, C_2)$. Hence, the algorithm immediately halts: because the analysis always comes out the same, as shown previously in Figure 6 and here in Figure 7d.

Efficiently dealing with interruptions. If, while processing a UC edge E, back-propagation encounters an as-yet-unprocessed UC edge E', the RUL^- algorithm interrupts its processing of E to deal with E'. Once it has generated all bypass edges for E', it updates the potential function and restarts its processing of E *from scratch.* Additional interruptions may require additional restarts. In all, this leads to a maximum of $2k$ processings of UC edges (including restarts).

Like Morris' 2014 algorithm, our new algorithm is recursive, which facilitates keeping track of information accumulated prior to interruptions so that, once interruptions are finished, processing can resume from where it left off—even if the potential function has been updated in the interim.

Figure 8 illustrates the approach using the sample STNU, but starting by attempting to process the UC edge $(C_2, C_2{:}{-10}, A_2)$ first, as shown in Figure 8a. In this case, an interruption is encountered at A_1. The relevant feature is that the distance from A_1 to C_2 is 0 which, being less than Δ_{C_2}, would warrant further back-propagation; however, back-propagation is not allowed past an unstarted UC edge. Therefore, the processing of $(C_2, C_2{:}{-10}, A_2)$ is interrupted. However, the accumulated information (i.e., the distance values shown on the righthand side) is not thrown away. After $(C_1, C_1{:}{-3}, A_1)$ has been fully processed, as seen before in Figure 7a, and the potential function has been updated, the processing of $(C_2, C_2{:}{-10}, A_2)$ resumes. First, the queue for the Dijkstra back-propagation is initialized with just A_1. The key for A_1 in the queue is computed using the *updated* potential function. (If there had been multiple interruptions, the queue would have been initialized with the ATPs for all the interrupting UC edges.) Figure 8b shows how back-propagation continues from A_1. When X is visited, it reveals a shorter distance to C_2; when C_2 is visited, a loop from C_2 to C_2 of length $5 < \Delta_{C_2}$ is detected which would trigger a separate forward propagation that, as seen before, would show the network to be non-DC.

In general, recursion enables handling interruptions more efficiently, avoiding redundant propagations incurred by the

Algorithm 1: The `UpdPF` function

```
Input: 𝒢, an STNU graph; A, an activation time-pt.; h, a
       pot. func. for 𝒢_ℓo, excluding edges ending at A
Output: A potential function h' for 𝒢_ℓo (including edges
        terminating at A); or ⊥ if 𝒢_ℓo is inconsistent
1  h' := copy-vector(h)
2  𝒬 := new empty priority queue
3  𝒬.insert(A, 0)              // Init. queue for back-prop from A
4  while (!𝒬.empty()) do
5      (V, key(V)) := 𝒬.extractMinNode()
       // Back-prop along ordinary edges ending at V
6      foreach ((U, δ, V) ∈ ℰ_o) do
7          if (UpdVal((U, δ, V), h, h', 𝒬)==⊥) then return ⊥
       // Back-prop along LC-edge ending at V, if any
8      if (V is contingent) then
9          (A_V, x_V, y_V, V) := contingent link for V
10         if (UpdVal((A_V, x_V, V), h, h', 𝒬) == ⊥) then
               return ⊥
11 return h'
```

Algorithm 2: `UpdVal`, helper for `UpdPF`

```
Input: h, h', potential functions; (U, δ, V), an edge; and 𝒬, a
       priority queue
Output: ⊤ iff h' can be updated to satisfy (U, δ, V) without
        detecting a negative loop
1 if (h'(U) < h'(V) − δ) then
2     h'(U) := h'(V) − δ
3     if (𝒬.state(U) == notYetInQ) then
4         𝒬.insert(U, h(U) − h'(U))
5     else if (𝒬.state(U) == inQ) then
6         𝒬.decreaseKey(U, h(U) − h'(U))
7     else return ⊥
8 return ⊤
```

RUL^- algorithm each time it restarts from scratch. In addition, our implementation can detect multiple interruptions, thereby reducing the number of resumptions. Finally, our algorithm does not insert any bypass edges for E until *after all interruptions have completed,* resulting in fewer redundant edge insertions and fewer updates to the potential function.

Pseudocode

Pseudocode for the RUL2021 algorithm is given as Algorithms 1 to 7, discussed in detail below.

Updating the Potential Function. Like the RUL^- algorithm, whenever new bypass edges are inserted into the LO-graph, the RUL2021 algorithm incrementally updates the potential function for the LO-graph using a propagation-based algorithm similar to that of Ramalingam et al. (1999). Pseudocode for the `UpdPF` (i.e., update potential) function is given as Algorithm 1. Its inputs include an activation time-point A, and a function h that satisfies all the edges in the LO-graph, *except* possibly edges ending at A (e.g., new bypass edges). It aims to return a function h' that satisfies *all* of the LO-edges, including those terminating at A.

Initially, h' is the same as h. `UpdPF` then propagates backward from A along LO-edges, adjusting the values of h' as necessary to satisfy the edges it encounters. It uses a priority queue $\mathcal{Q}$ where the key for each node X reflects the amount by which $h'(X)$ had to be changed. Propagation ceases at nodes for which no adjustment of h' is necessary.

For each encountered edge (U, δ, V), the `UpdVal` (i.e., update value) function in Algorithm 2 adjusts $h'(U)$ if necessary to ensure that it satisfies the constraint $V - U \leq \delta$. If U is not yet in the queue, it inserts U into the queue; if U is already in the queue, it decreases the key for U to reflect the new value of $h'(U)$; but if U has already been popped from the queue, it immediately returns $\perp$, since this can only happen if this edge completed a negative cycle back to A.

Phase I. The first phase of processing a UC-edge involves propagating backward from its contingent time-point C, along edges in the LO-graph, looking for opportunities to apply the R and L rules from Table 2. The `phase1` function in Algorithm 3 uses a potential function to re-weight the edges in the LO-graph to enable a (single-sink) Dijkstra-like traversal of the shortest paths. The key for each node X in the priority queue is the distance from X to C in the re-weighted graph, while δ_{xc} is the corresponding distance in the original LO-graph. When X is popped from the queue, significant processing only continues if $\delta_{xc} < \Delta_C$. (For now, ignore the `loc.dist[X]` variable.) The first three cases are those in which back-propagation does *not* continue past X.

In Case 1, $X \equiv C$. If $\delta_{xc} < 0$, then a negative cycle has been found and $\perp$ is immediately returned. Otherwise, a flag `loc.ccLoop` is set, indicating that a non-negative cycle from C back to C has been found. (More on this later.)

In Case 2, X is the ATP for a UC-edge E_X that the algorithm has not yet started processing. For now, that UC-edge is accumulated in `loc.UUCEs` (i.e., *unstarted* UC-edges).

In Case 3, X is the ATP for a UC-edge E_X that the algorithm has started processing, but not yet finished. In other words, a cycle of interrupted processings of UC edges has been detected. Thus, the function immediately returns $\perp$.

Case 4 is where back-propagation from X continues. The `apRL` (i.e., apply relax/lower) function in Algorithm 4 considers each LO-edge $e = (W, \theta, X)$ incoming to X and accumulates (but does not insert) a new edge from W to C by applying the R or L rule to e and (X, δ_{xc}, C). The edges (W, δ_{wc}, C) returned by `apRL` are then used in Case 4 to either insert W into the priority queue or decrease its key if it is already in the queue. When W is eventually popped off the queue, back-propagation from W will then be considered.

The RUL2021 algorithm. Algorithm 5 gives the high-level structure of the RUL2021 algorithm. It takes an STNU graph $\mathcal{G}$ as its input. It outputs $\top$ if and only if $\mathcal{G}$ is DC. It first initializes a *global* data structure `glo`, whose fields are `pf` and `status`. The `pf` field is initialized to a potential function for the LO-graph obtained from a call to Bellman-Ford. The `status` field is a vector that holds the processing status of each UC-edge. The initial status of each UC-edge is `nYet` (i.e., not yet started). It then calls the recursive helper function `RULbp` (Algorithm 6) on each UC-edge.

The `RULbp` (RUL back-prop) function first checks the status of the input UC-edge E (Lines 3 to 4). If E is already started, then a negative cycle of recursive calls has been detected, and `RULbp` returns $\perp$. Alternatively, if E has already

Algorithm 3: The `phase1` function

```
Input: 𝒢 = (𝒯, ℰ), an STNU graph; C ∈ 𝒯_C, a contingent TP;
       𝒬, a priority queue; glo, instance of global struct; loc,
       instance of local struct
Output: ⊥, iff back-prop. from C reveals 𝒢 to be not DC
// Side Effect: Modifies contents of loc struct
1  h := glo.pf      // Potential fn., a solution to the LO-graph
2  st := glo.status          // Gives status of UC-edges
3  loc.UUCEs := {}       // Collects unstarted UC-edges
4  while not(𝒬.empty()) do
       // key(X) = dist. from X to C, adjusted using pot. fn.
5      (X, key(X)) := 𝒬.extractMinNode()
6      δ_xc := key(X) − h(X)             // δ_xc = XC dist. in 𝒢
       // If X is an ATP, then E_X is corresp. UC-edge; else ⊥
7      E_X := 𝒢.UCEdgeFromATP(X)
8      if (δ_xc < loc.dist[X]) then
9          loc.dist[X] := δ_xc        // Record shorter length
10         if (δ_xc < Δ_C) then                // Back-propagate
11             if (X ≡ C) then
                   // Case 1: CC loop of length δ_xc < Δ_C
12                 if (δ_xc < 0) then return ⊥      // Neg. cycle
13                 else loc.ccLoop := ⊤     // Alert fwd. prop.
14             else if (E_X and st[E_X] == nYet) then
                   // Case 2: E_X is an unstarted UC-edge
15                 loc.UUCEs.add((E_X, X))
16             else if (E_X and st[E_X] == std) then
                   // Case 3: Cycle of interruptions: not DC
17                 return ⊥
18             else
                   // Case 4: Back-prop. along LO-edges
19                 foreach (W, δ_WC) ∈ apRL(𝒢, X, Δ_C, δ_xc) do
20                     if δ_WC < 𝒢.ordEdgeWt(W, C) then
21                         newKey := δ_WC + h(W)
22                         𝒬.insOrDecrKey(W, newKey)
```

Algorithm 4: The `apRL` algorithm

```
Input: 𝒢, an STNU graph; V ∈ 𝒯_X; Δ_C; δ_VC
Output: A list of pairs, (W, δ_WC), obtained by applying the R
        and L rules to LO-edges incoming to V, with the edge
        (V, δ_VC, C).
1  edges := {}
2  if (δ_VC ≥ Δ_C) then return {}         // RL rules don't apply
3  if (V ∈ 𝒯_C) then
       // Apply the L rule to (A_V, v:x_V, V) and (V, δ_VC, C)
4      edges.add((A_V, x_V + δ_VC))
5  else
6      foreach ((W, δ_WV, V) ∈ ℰ_o) do
           // Apply R rule to (W, δ_WV, V) and (V, δ_VC, C)
7          edges.add((W, δ_WV + δ_VC))
8  return edges
```

Algorithm 5: The RUL2021 algorithm

```
Input: 𝒢 = (𝒯, ℰ = ℰ_o ∪ ℰ_ℓ ∪ ℰ_u), an STNU graph
Output: ⊤, if 𝒢 is DC; ⊥, otherwise.                // Modifies 𝒢
1  glo := new global struct              // Fields: pf, status
2  glo.pf := BellmanFord (𝒢_ℓo)
3  if (glo.pf == ⊥) then return ⊥
   // Initialize processing status of each upper-case edge
4  glo.status := [nYet,..., nYet]                    // k-vector
5  foreach (E = (C, C:−y, A) ∈ ℰ_u) do
6      if (RULbp (𝒢, E, glo) == ⊥) then return ⊥
7  return ⊤
```

been fully processed, then `RULbp` returns $\top$. The rest of `RULbp` deals with the two-phase processing of E.

First, a *local* data structure `loc` is initialized. Its `ccLoop` field is set to $\bot$ because no loop from C back to C has yet been found; and its `dist` vector is initialized so that $\texttt{dist}[X] = \infty$ for each X, representing that no path from any X to C has yet been explored. Next (Lines 10 to 11) a priority queue $\mathcal{Q}$ is initialized to include each X for which there is an ordinary edge (X, δ_{xc}, C), terminating at C. Back propagation during phase one will begin at one of these Xs.

Each iteration of the `while` loop (Lines 13 to 22) attempts to start/re-start processing the UC-edge E by calling the `phase1` function (Line 14). After that (Line 15), the `loc.UUCEs` field is checked to see whether any unstarted UC-edges were encountered by `phase1`. If so, processing of E is interrupted to allow all of those as-yet-unstarted UC-edges to be processed (Line 17). Any of these may, in turn, result in further recursive calls to `RULbp`. If any of these interrupting calls fail, then `RULbp` immediately returns $\bot$. Otherwise, at Lines 18 to 20, it prepares the queue for re-starting the phase-one processing of E in the next iteration of the `while` loop. After clearing the queue, the activation time-point for each interrupting UC-edge is re-inserted into the queue with a key based on the potential function that quite likely was updated during the recursive call(s) to `RULbp` (Line 20). Those ATPs provide starting points for resuming the phase-one processing of E. The `dist` values for those ATPs are set to ∞ to ensure that when they are popped off the queue by `phase1` they will satisfy the conditional at Line 8 in `phase1`. In contrast, the `dist` values for all other time-points are preserved so that the phase-one processing of E does not need to restart from scratch.

Eventually, some iteration of the `while` loop must result in either detecting a cycle of recursive calls and returning $\bot$ or successfully completing the phase-one processing of E. In the latter case, the `loc.ccLoop` flag is checked (Line 23) to see whether the processing of E detected a *non-negative* CC loop. If so, the `fwdPropNDC` function, described below, does a separate forward propagation from C to determine whether the *LC-edge* $(A, c{:}x, C)$ can be reduced away and, if it can, cause RUL2021 to return $\bot$. Otherwise, phase two is carried out (Lines 23 to 28). It generates and *inserts* any bypass edges derived from data in the `loc.dist` vector—but only for cases in which the *length-preserving* $\mathrm{U_{lp}}$ rule applies. If any new edges are inserted, the potential function is updated (Line 30) and the status of E is set to `done`.

Separate forward propagation. The `fwdPropNDC` function (Algorithm 7) takes as input information about the phase-one processing of a UC-edge $(C, C{:}{-y}, A)$ in which a CC loop L was encountered with $0 \leq L < \Delta_C$. It does a forward propagation along paths in the LO-graph emanating

Algorithm 6: The $\texttt{RULbp}$ algorithm

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, STNU graph; $\mathbf{E} \in \mathcal{E}_u$; $\texttt{glo}$
Output: $\top$, iff $\mathbf{E}$ can be successfully reduced away

1 $\texttt{st} := \texttt{glo.status}$ // Gives status of UC-edges
2 $h := \texttt{glo.pf}$ // Potential function for LO-graph
3 **if** $(\texttt{st}(\mathbf{E}) == \texttt{std})$ **then return** $\bot$ // Negative cycle
4 **if** $(\texttt{st}(\mathbf{E}) == \texttt{done})$ **then return** $\top$ // $\mathbf{E}$ already done
5 $\texttt{st}(\mathbf{E}) := \texttt{std}$ // Set status of $\mathbf{E}$ to *started*
6 $\Delta_C := y - x$ // $\mathbf{E}$ is UC-edge for cont. link (A, x, y, C)
7 $\texttt{loc}$:= new *loc* struct // fields: $\texttt{ccLoop}$, $\texttt{dist}$, $\texttt{UUCEs}$
8 $\texttt{loc.ccLoop} := \bot$ // No CC loop found yet
9 $\texttt{loc.dist} := [\infty, \ldots, \infty]$ // n vector: dist from TPs to C
10 $\mathcal{Q}$:= a new priority queue (initialized below)
// $key(X) = h(X) + \delta_{xc}$ = adjusted distance from X to C
11 **foreach** $((X, \delta_{xc}, C) \in \mathcal{E}_o)$ **do** $\mathcal{Q}.ins(X, h(X) + \delta_{xc})$
12 $\texttt{continue?} := \top$
13 **while** $(\texttt{continue?})$ **do**
14   **if** ! $(\texttt{phase1}(\mathcal{G}, C, \mathcal{Q}, \texttt{glo}, \texttt{loc}))$ **then return** $\bot$
15   **if** $\texttt{loc.UUCEs} != \{\}$ **then**
        // Proc. unstarted UC-edges seen by $\texttt{phase1}$
16   **foreach** $(\mathbf{E}_X, X) \in \texttt{loc.UUCEs}$ **do**
17   **if** ! $(\texttt{RULbp}(\mathcal{G}, \mathbf{E}_X, \texttt{glo}))$ **then return** $\bot$
18         $\mathcal{Q}.clear()$ // Prep. $\mathcal{Q}$ for next iteration of WHILE
19   **foreach** $((\mathbf{E}_X, X) \in \texttt{loc.UUCEs})$ **do**
20             $\mathcal{Q}.ins(X, \texttt{loc.dist}[X] + \texttt{glo.pf}[X])$
21             $\texttt{loc.dist}[X] := \infty$
22   **else** $\texttt{continue?} := \bot$
23 **if** $(\texttt{loc.ccLoop}$ **and** $\texttt{fwdPropNDC}(\mathcal{G}, C, \Delta_C, \texttt{loc.dist}, \texttt{glo.pf}))$ **then return** $\bot$
// Collect edges from applying U_{lp} rule to XC edges
24 $\texttt{edges?} := \bot$
25 **foreach** $(X \in \mathcal{T}$ *such that* $X \not\equiv C)$ **do**
26     $\delta_{xc} := \texttt{loc.dist}[X]$
27   **if** $(\Delta_C \leq \delta_{xc} < \infty)$ **then** // Length-preserving case
28         $\mathcal{G}.insOrUpdateOrdEdge(X, \delta_{xc} - y, A)$
29         $\texttt{edges?} := \top$
30 **if** $(\texttt{edges?})$ **then** $\texttt{glo.pf} := \texttt{UpdPF}(\mathcal{G}, A, \texttt{glo.pf})$
31 **if** $(\texttt{glo.pf} == \bot)$ **then return** $\bot$
32 $\texttt{st}[\mathbf{E}] := \texttt{done}$
33 **return** $\top$ // Processing of E completed

from the contingent time-point C looking for opportunities to reduce away the corresponding LC-edge $(A, c{:}x, C)$. Intuitively, if any time-point X in that CC loop is constrained to occur *before* C, the network must not be DC, since the loop has less flexibility than the contingent link. The forward propagation restricts attention to time-points X for which $\texttt{loc.dist}[X] < \Delta_C$. It uses the potential function to re-weight the LO-edges so that Dijkstra can be used to guide the exploration of paths. If it ever finds a path from C to some X of negative length, it immediately returns $\top$, indicating that the LC-edge can be reduced away.

Experimental Evaluation

The performance of our new RUL2021 DC-checking algorithm was compared against that of the pre-existing RUL^- and Morris 2014 algorithms. All algorithms/procedures were implemented in Java and run on a JVM 11 having 8GB of

Algorithm 7: $\texttt{fwdPropNDC}$

Input: $\mathcal{G}$, an STNU graph; $C \in \mathcal{T}_C$; $\Delta_C = y - x$; $\texttt{dist}$, vector of XC distances; h, potential function
Output: $\top$, iff LC-edge $(A, c{:}x, C)$ can be reduced away

1 $\mathcal{Q}$:= new priority queue // Key$[X] = d(C, X) - h(C)$
2 $\mathcal{Q}.insert(C, -h(C))$ // Queue initially contains only C
3 **while** $(!\mathcal{Q}.empty())$ **do**
4     $(X, key(X)) := \mathcal{Q}.extractMinNode()$
5     $d(C, X) := key(X) + h(X)$ // CX distance in $\mathcal{G}_{\ell o}$
6   **if** $(\texttt{dist}[X] < \Delta_C)$ **then** // Only visit these Xs
        // Check if CX path *can* reduce-away the LC-edge
7   **if** $(d(C, X) < 0)$ **then return** $\top$
        // Else iterate over edges emanating from X
8   **foreach** $((X, \delta_{XY}, Y) \in \mathcal{E}_\ell \cup \mathcal{E}_o)$ **do**
9             $\texttt{newKey} := d(C, X) + \delta_{XY} - h(Y)$
10             $\mathcal{Q}.insOrDecrKey(Y, \texttt{newKey})$
11 **return** $\bot$ // Was unable to reduce-away the LC-edge

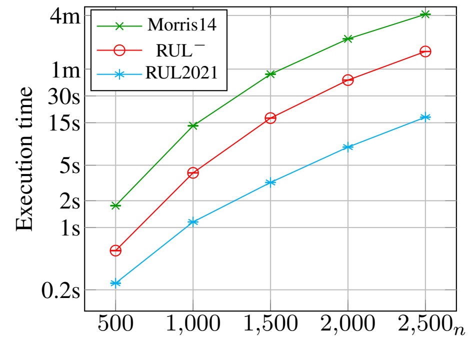


Figure 9: Test 1: Execution time vs. n; DC networks.

heap memory on a Linux box with one Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz. The implementations are freely available as a Java library (Posenato 2022).

Due to space limitations, we present here a summary of the tests and results. Full details are available from Hunsberger and Posenato (2021).

We set up a generator that randomly generates realistic STNUs with a chosen topology, tunable by multiple parameters. The benchmarks are available from Posenato (2022). Because DC STNUs tend to be more challenging for DC-checking algorithms, this section focuses on DC STNUs.

Test 1: Execution-time vs. number of nodes. For each $n \in \{500, 1000, 1500, 2000, 2500\}$, we randomly generated 200 DC and 200 non-DC STNUs, each having n nodes, $k = n/10$ contingent links, and $m \approx 6n$ edges.

Figure 9 displays the average execution times of the algorithms for the DC networks. Each plotted point represents the average execution time for an algorithm on the 200 instances of the given size; each error bar shows the 95% confidence interval. These results demonstrate that RUL2021 performs significantly better (by an order of magnitude) than the other algorithms over DC instances. Results for non-DC networks were similar.

One of our principal motivating hypotheses was that our new algorithm would be significantly faster than the RUL^-

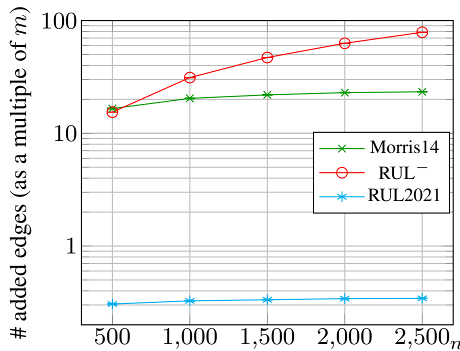


Figure 10: Test 1: Number of added edges (as a multiple of m) vs. the number of nodes in DC networks.

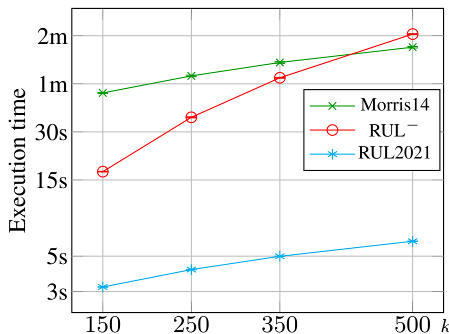


Figure 11: Test 2: Execution time vs. number of contingent links, k, in DC networks with 1500 nodes.

algorithm because it inserts significantly fewer new edges into the STNU graph.

Figure 10 dramatically confirms this hypothesis. It shows the number of edges inserted by each algorithm as a multiple of the number of edges m in the input graph, using a logarithmic scale. For DC networks, our algorithm inserted, on average, fewer than $0.4m$ new edges, while the other algorithms inserted $20m$ to $80m$ edges. Although avoiding the non-length-preserving U_{nlp} rule sometimes requires our algorithm to do extra forward propagations, this cost is far outweighed by the benefit of inserting so few new edges. Results for non-DC networks were similar.

Test 2: Execution-time vs. the number of contingent links. For this test, we generated random STNUs, each having 1500 nodes and between 150 and 500 contingent links. We built eight new benchmarks—four with 200 DC instances each, and four with 200 non-DC instances each—where $k \in \{150, 250, 350, 500\}$.

Figure 11 plots the execution times of the three algorithms across these new benchmarks for the DC instances. It confirms that the RUL2021 algorithm is fastest even when the number of contingent links increases. It also shows that the execution time of the Morris14 algorithm increases more slowly than that of the RUL$^-$ algorithm as the number of contingent links increases until, eventually, the Morris14 algorithm runs faster than the RUL$^-$ algorithm (when there are 500 contingent links). We believe this occurs because: (1) just as the RUL2021 algorithm only inserts edges needed to reduce away upper-case edges, the Morris14 algorithm only inserts edges needed to reduce away negative edges; and (2) as the number of contingent links increases, the numbers of negative nodes and contingent links converge, hence the max number of times the main processing functions of the two algorithms are called also converge. These conjectures are supported by extra plots in (Hunsberger and Posenato 2021), including cases where $k \approx \sqrt{n}$.

Summary. The RUL2021 algorithm achieves a dramatic improvement over the performance of the RUL^- algorithm on STNU graphs by (1) inserting fewer edges into the STNU graph to speed up the many instances of Dijkstra-like traversals; and (2) avoiding redundant computations when the processing of one UC edge is interrupted by one or more other UC edges. The first goal was achieved by, first, only computing *path-lengths* associated with applications of the L and R rules while refraining from inserting any new edges associated with those paths and, second, only using the *length-preserving* case of the U rule. Although avoiding the *non-length-preserving* case occasionally requires performing separate forward propagations, the savings from not inserting so many edges far outweighs the cost of those rare forward propagations. The second goal was achieved by implementing the processing of UC edges recursively, not inserting any new edges until all recursive interruptions are completed, and keeping track of work done prior to interruptions so that propagation can continue from where it left off, even if the potential function has been updated in the interim.

We conjecture that our RUL2021 algorithm provides the most improvement over the RUL^- algorithm in networks with multiple pathways of relatively small non-negative weights terminating in contingent time-points (i.e., in networks where tasks with uncertain durations face multiple short-range deadlines arising from interactions with other tasks). If there are only long-range (e.g., global) deadlines, then the performance of the algorithms may be closer.

Conclusions

The STNU model is an established formalism for representing and reasoning about time. The most important problem associated with STNUs is the DC-checking problem (i.e., checking whether a given STNU is dynamically controllable). The RUL$^-$ DC-checking algorithm has the best worst-case time-complexity: $O(mn + k^2n + kn \log n)$. We believe that this theoretical complexity cannot be improved upon. However, our empirical evaluation demonstrates that the performance of our new RUL2021 algorithm can be up to *ten times faster*. This improved performance is expected to increase the practicality of STNUs for real-world applications.

All implementations and benchmarks used for our evaluation are freely available at (Posenato 2022).

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1909739.

Experiments supported by Centro Piattaforme Tecnologiche (CPT) at University of Verona, Italy.

References

Bhargava, N.; and Williams, B. C. 2019a. Faster Dynamic Controllability Checking in Temporal Networks with Integer Bounds. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, 5509–5515. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-4-1.

Bhargava, N.; and Williams, B. C. 2019b. Faster Dynamic Controllability Checking in Temporal Networks with Integer Bounds. http://mers-papers.csail.mit.edu/Conference/2019/IJCAI_2019_Bhargava/ijcai19.pdf. Accessed: 2021-12-07.

Cairo, M.; Hunsberger, L.; and Rizzi, R. 2018. 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*, 8:1–8:16.

Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; and Stein, C. 2009. *Introduction to Algorithms*. The MIT Press, 3rd edition.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence*, 49(1-3): 61–95.

Franceschetti, M.; and Eder, J. 2019. Computing Ranges for Temporal Parameters of Composed Web Services. In *21st International Conference on Information Integration and Web-based Applications & Services*, iiWAS2019, 537–545. ISBN 978-1-4503-7179-7.

Franceschetti, M.; and Eder, J. 2020a. Designing Decentralized Business Processes with Temporal Constraints. In *Advanced Information Systems Engineering*, 51–63. ISBN 978-3-030-58135-0.

Franceschetti, M.; and Eder, J. 2020b. Negotiating Temporal Commitments in Cross-Organizational Business Processes. In *27th International Symposium on Temporal Representation and Reasoning (TIME-2020)*, volume 178 of *Leibniz International Proceedings in Informatics (LIPIcs)*, 4:1–4:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Hunsberger, L. 2015. Efficient Execution of Dynamically Controllable Simple Temporal Networks with Uncertainty. *Acta Informatica*, 53(2): 89–147.

Hunsberger, L.; and Posenato, R. 2021. A note on speeding up DC-checking for STNUs. https://iris.univr.it/handle/11562/1045707. Accessed: 2022-03-07.

Karpas, E.; Levine, S. J.; Yu, P.; and Williams, B. C. 2015. Robust Execution of Plans for Human-Robot Teams. In *25th Int. Conf. on Automated Planning and Scheduling (ICAPS-15)*, volume 25, 342–346.

Morris, P. 2006. A Structural Characterization of Temporal Dynamic Controllability. In *Principles and Practice of Constraint Programming (CP-2006)*, volume 4204, 375–389.

Morris, P. 2014. Dynamic controllability and dispatchability relationships. In *CPAIOR 2014*, volume 8451 of *LNCS*, 464–479. Springer.

Morris, P. H.; and Muscettola, N. 2005. Temporal Dynamic Controllability Revisited. In *20th National Conference on Artificial Intelligence (AAAI-2005)*, 1193–1198.

Morris, P. H.; Muscettola, N.; and Vidal, T. 2001. Dynamic Control Of Plans With Temporal Uncertainty. In *17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, 494–502.

Posenato, R. 2022. CSTNU Tool: A Java library for checking temporal networks. *SoftwareX*, 17: 100905.

Ramalingam, G.; Song, J.; Joskowicz, L.; and Miller, R. E. 1999. Solving Systems of Difference Constraints Incrementally. *Algorithmica*, 23(3): 261–275.