Faster Dynamic Controllability Checking for Simple Temporal Networks with Uncertainty

Massimo Cairo
Department of Mathematics, University of Trento, Italy
massimo.cairo@unitn.it

Luke Hunsberger
Computer Science Department, Vassar College, Poughkeepsie, NY USA
hunsberger@vassar.edu

Romeo Rizzi
Department of Computer Science, University of Verona, Italy
romeo.rizzi@univr.it

Abstract

Simple Temporal Networks (STNs) are a well-studied model for representing and reasoning about time. An STN comprises a set of real-valued variables called time-points, together with a set of binary constraints, each of the form $Y \leq X + w$. The problem of finding a feasible schedule (i.e., an assignment of real numbers to time-points such that all of the constraints are satisfied) is equivalent to the Single Source Shortest Path problem (SSSP) in the STN graph.

Simple Temporal Networks with Uncertainty (STNUs) augment STNs to include *contingent links* that can be used, for example, to represent actions with uncertain durations. The duration of a contingent link is not controlled by the planner, but is instead controlled by a (possibly adversarial) environment. Each contingent link has the form, $\langle A, \ell, u, C\rangle$, where $0 < \ell \leq u < \infty$. Once the planner executes the activation time-point A, the environment must execute the contingent time-point C at some time $A + \Delta$, where $\Delta \in [\ell, u]$. Crucially, the planner does not know the value of Δ in advance, but only discovers it when C executes. An STNU is dynamically controllable (DC) if there is a strategy that the planner can use to execute all of the non-contingent time-points, such that all of the constraints are guaranteed to be satisfied no matter which durations the environment chooses for the contingent links. The strategy can be dynamic in that it can react in real time to the contingent durations it observes. Recently, an upper bound of $O(N^3)$ was given for the DC-checking problem for STNUs, where N is the number of time-points.

This paper introduces a new algorithm, called the RUL^- algorithm, for solving the DC-checking problem for STNUs that improves on the $O(N^3)$ bound. The worst-case complexity of the RUL^- algorithm 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 time-points. If M is $O(N^2)$, then the complexity reduces to $O(N^3)$; however, in sparse graphs the complexity can be much less. For example, if M is $O(N \log N)$, and K is $O(\sqrt{N})$, then the complexity of the RUL^- algorithm reduces to $O(N^2 \log N)$.

The RUL^- algorithm begins by using the Bellman-Ford algorithm to compute a potential function. It then performs at most $2K$ rounds of computations, interleaving novel applications of Dijkstra's algorithm to (1) generate new edges and (2) update the potential function in response to those new edges. The constraint-propagation/edge-generation rules used by the RUL^- algorithm are distinguished from related work in two ways. First, they only generate *unlabeled* edges. Second, their applicability conditions are more restrictive. As a result, the RUL^- algorithm requires only $O(K)$ rounds of Dijkstra's algorithm, instead of the $O(N)$ rounds required by other approaches. The paper proves that the RUL^- algorithm is sound and complete for the DC-checking problem for STNUs.

2012 ACM Subject Classification Networks → Network algorithms

Keywords and phrases Simple Temporal Networks with Uncertainty, Dynamic Controllability, Temporal Planning under Uncertainty

Digital Object Identifier 10.4230/LIPIcs.TIME.2018.8

1 Introduction

Simple Temporal Networks (STNs) are a well-studied model for representing temporal constraints [5]. An STN comprises a set of time-points $\{P, Q, R, \dots\}$, together with a set of constraints on those time-points, where each constraint has the form $Q \leq P + w$ with $w \in \mathbb{R}$. The goal for a planning agent is to schedule the execution of the time-points (i.e., to assign a real value to each variable $P, Q, R, \dots$ representing its execution time) so that all of the constraints in the network are satisfied. The STN model forms the base of many other models for temporal reasoning and planning.

It is possible to check whether an STN is consistent (i.e., whether it admits a schedule that satisfies all of the constraints, and find such a scheduling if one exists) in polynomial time. Specifically, this problem can be reduced to the Single Source Shortest Path problem for the STN graph [5], which contains a node for each time-point and a weighted, directed edge for each constraint. In turn, this problem can be solved with the Bellman-Ford algorithm [4], whose running time is $O(MN)$, where N is the number of time-points, and M is the number of edges in the network.

Simple Temporal Networks with Uncertainty (STNUs) extend STNs to include *contingent links,* which can be used, for example, to represent actions with uncertain durations [14]. Each contingent link has the form $\langle A, \ell, u, C\rangle$, where A is called the *activation* time-point, C is called the *contingent* time-point, and $0 < \ell \leq u < \infty$. The contingent link is activated when the time-point A is executed. Once that happens, the execution of C is determined not by the planning agent, but by the (possibly adversarial) environment. In particular, the environment must execute C at some time Δ after the execution of A, where $\Delta \in [\ell, u]$. The value Δ is called the duration of the contingent link, it is under the control of the environment, and is unknown to the planner until C is actually executed.

An STNU is said to be dynamically controllable (DC) if there exists a strategy for the planning agent to execute all of the *non-contingent* (a.k.a., *executable*) time-points such that all of the constraints in the network are guaranteed to be satisfied no matter how the durations of the contingent links are chosen by the environment. The strategy must be *dynamic* in the sense that the execution time it chooses for each executable time-point X can only depend on the durations of contingent links that have already completed. In other words, the strategy's execution decisions must depend only on past execution events.

The DC-checking problem for STNUs, hereinafter called the STNU-DC problem, is that of determining whether any given STNU is DC. Recently, Morris [12] presented an $O(N^3)$-time algorithm for the STNU-DC problem.

Being such a simple and flexible model, STNUs have been extended in several ways in the literature [15, 10, 3, 6, 2]. For this reason, it is crucial to study and optimize algorithms for the STNU-DC problem.

2 Preliminaries and notation

Following Morris et al. [14], an STNU is a tuple $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ where:

- $\mathcal{T}$ is a finite set of real-valued variables called *time-points,* denoted by capital letters $P, Q, R, \dots$;

- $\mathcal{C}$ is a finite set of *constraints,* each of the form $Q \leq P + w$, for some $P, Q \in \mathcal{T}$ and $w \in \mathbb{R}$; and
- $\mathcal{L}$ is a finite set of *contingent links,* each of the form $\langle A, \ell, u, C\rangle$, for some $A, C \in \mathcal{T}$ and $\ell, u \in \mathbb{R}$, where $0 < \ell \leq u < \infty$.

In a contingent link $\langle A, \ell, u, C\rangle$, A is called the *activation* time-point, and C is called the *contingent* time-point. Distinct contingent links must have distinct contingent time-points. Given the contingent time-point C of a contingent link $\langle A, \ell, u, C\rangle \in \mathcal{L}$ we define $A^C = A$, $u^C = u$, $l^C = l$; however, when context allows, the superscripts will be omitted. The set of contingent time-points is $\mathcal{T}_\mathrm{C} \coloneqq \{C \mid (A, l, u, C) \in \mathcal{L}\}$; and the set of *executable* (or *non-contingent*) time-points is $\mathcal{T}_\mathrm{X} = \mathcal{T} \setminus \mathcal{T}_\mathrm{C}$. As in prior work, and without loss of generality, we assume that each activation time-point is an executable time-point. In the following, we assume that an STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ is given.

2.1 Dynamic controllability

Following Morris et al. [14], a *situation* is a function $\sigma\colon \mathcal{T}_\mathrm{C} \to \mathbb{R}$ that assigns a *duration* $\Delta^C_\omega \coloneqq \omega(C) \in [\ell, u]$ to each contingent link $\langle A, \ell, u, C\rangle \in \mathcal{L}$. The set of all possible situations is denoted by $\Omega \coloneqq \prod_{C \in \mathcal{T}_\mathrm{C}} [\ell^C, u^C]$. An *execution strategy* is a function $\sigma\colon (\Omega, \mathcal{T}_\mathrm{X}) \to \mathbb{R}$ that assigns an *execution time* $[\sigma(\omega)]_X \coloneqq \sigma(\omega, X)$ to each executable time-point X, in each possible situation $\omega \in \Omega$. Given a contingent link $\langle A, \ell, u, C\rangle \in \mathcal{L}$, we define the *execution time* of its contingent time-point C to be $[\sigma(\omega)]_C \coloneqq [\sigma(\omega)]_A + \Delta^C_\omega$. In general, $[\sigma(\omega)]_P \in \mathbb{R}$ denotes the execution time of the (executable or contingent) time-point $P \in \mathcal{T}$ in the situation ω under the strategy σ.

An execution strategy σ is *viable* if it satisfies every constraint (i.e., for each constraint $Q \leq P + w$ in $\mathcal{C}$, and each situation $\omega \in \Omega$, $[\sigma(\omega)]_Q \leq [\sigma(\omega)]_P + w$). The strategy σ is *dynamic* if, for any two situations $\omega_1, \omega_2 \in \Omega$, and for each executable time-point $X \in \mathcal{T}_\mathrm{X}$, if $\{\langle C, \Delta^C_{\omega_1}\rangle \mid [\sigma(\omega_1)]_C < [\sigma(\omega_1)]_X\} = \{\langle C, \Delta^C_{\omega_2}\rangle \mid [\sigma(\omega_2)]_C < [\sigma(\omega_1)]_X\}$ then $[\sigma(\omega_2)]_X = [\sigma(\omega_1)]_X$. This definition correctly captures the intuitive notion that the execution time of X can only depend only on the durations of contingent links whose contingent time-points have executed *before* X [7]. An STNU is said to be *dynamically controllable* (DC) if it admits an execution strategy that is both dynamic and viable.

To simplify the mathematics, Morris [11] provided an alternative notion of dynamic controllability that allows a dynamic strategy to react *instantaneously* to observations of contingent executions. The only change required to the above definition of a dynamic strategy is to replace "$<$" by "$\leq$". That change allows the execution of X to depend on contingent links that have completed *at or before* X. The rest of the paper presumes DC with instantaneous reaction.

2.2 Constraint Propagation/Edge Generation in STNU Graphs

Following Morris and Muscettola [13], each STNU can be represented by a directed, weighted graph where the nodes correspond to the time-points, and the edges come in three varieties depending on whether and how they are labeled. First, each constraint $Q \leq P + w$ in $\mathcal{C}$ is represented by an *ordinary* (i.e., unlabeled) edge from P to Q of length (or weight) w, notated as (P, w, Q).[1] Next, for each contingent link $\langle A, \ell, u, C\rangle$ there is a *lower-case* edge from A to C labeled by $[c\colon\ell]$ that represents the *uncontrollable possibility* that the contingent

[1] Putting the weight w between the time-points P and Q makes notating paths easier. For example, the path consisting of the consecutive edges (P, w, Q) and (Q, v, R) can be notated as (P, w, Q, v, R).

$$\begin{tikzcd} X & C_1 \arrow[l, "-1"'] \arrow[d, bend left, "{[C_1:-9]}"] & C_2 \arrow[l, "2"'] \arrow[d, bend left, "{[C_2:-7]}"] \\ & A_1 \arrow[u, bend left, "{[c_1:2]}"] & A_1 \arrow[u, bend left, "{[c_2:3]}"] \end{tikzcd}$$

Figure 1 A sample STNU graph with two contingent links $(A_1, 2, 9, C_1)$ and $(A_2, 3, 7, C_2)$.

Table 1 Contraint-propagation rules from Morris and Muscettola [13].

Rule	Graphical Representation	Applicability Conditions
No Case	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, dashed, bend right, "v+w"'] & Y \arrow[r, "w"] & W \end{tikzcd}$	(none)
Upper Case	$\begin{tikzcd} X \arrow[r, "v"] \arrow[rr, dashed, bend right, "{[C:v-w]}"'] & Y \arrow[r, "{[C:-w]}"] & A \end{tikzcd}$	(none)
Lower Case	$\begin{tikzcd} A \arrow[r, "{[c:\ell]}"] \arrow[rr, dashed, bend right, "\ell+w"'] & C \arrow[r, "w"] & X \end{tikzcd}$	$w < 0$
Cross Case	$\begin{tikzcd} A \arrow[r, "{[c:\ell]}"] \arrow[rr, dashed, bend right, "{[K:\ell-w]}"'] & C \arrow[r, "{[K:-w]}"] & A^K \end{tikzcd}$	$K \not\equiv C,\ w < 0$
Label Removal	$\begin{tikzcd} X \arrow[r, "{[C:-y]}"] \arrow[r, dashed, bend right, "-y"'] & A \arrow[r, "{[c:\ell]}"] & C \end{tikzcd}$	$-y \geq -\ell$

duration $C - A$ might take on its lower bound ℓ; and an *upper-case* edge from C to A labeled by $[C\!:\!-u]$ that represents the *uncontrollable possibility* that the contingent duration $C - A$ might take on its upper bound u. These edges are notated as $(A, [c\!:\!\ell], C)$ and $(C, [C\!:\!-u], A)$, respectively. The sets of ordinary, lower-case and upper-case edges are denoted by $\mathcal{E}^o, \mathcal{E}^\ell$ and $\mathcal{E}^u$, respectively:

- $\mathcal{E}^o = \{(P, w, Q) \mid (Q \leq P + w) \in \mathcal{C}\}$
- $\mathcal{E}^\ell = \{(A, [c\!:\!\ell], C) \mid \langle A, \ell, u, C\rangle \in \mathcal{T}_{\text{C}}\}$
- $\mathcal{E}^u = \{(C, [C\!:\!-u], A) \mid \langle A, \ell, u, C\rangle \in \mathcal{T}_{\text{C}}\}$

Fig. 1 shows a sample STNU graph borrowed from Hunsberger [8].

For convenience, the graph consisting of all of the ordinary and upper-case edges (i.e., $\mathcal{E}^o \cup \mathcal{E}^u$) is called the OU-graph; and the graph consisting of all of the lower-case and ordinary edges (i.e., $\mathcal{E}^o \cup \mathcal{E}^\ell$) is called the LO-graph.

Morris et al. [14] introduced a set of *triangular reductions* that formed the basis of a *pseudo-polynomial* DC-checking algorithm for STNUs. Their reductions generated and propagated a new kind of conditional constraint called a *wait* that captures part of the dynamism of the STNU-DC problem. Each wait constraint can be glossed as "while the contingent time-point C remains unexecuted, the time-point X must wait until $A + \delta$, where A is the activation time-point for C and $\delta \in \mathbb{R}$." Morris and Muscettola [13] then recast those reductions as the five constraint-propagation rules listed in Table 1.[2] In the figure, the pre-existing edges are drawn with solid arrows, while the generated edges are drawn with dashed arrows.

[2] The rules in Table 1 are presented in the form that allows instantaneous reaction. To disallow instantaneous reaction, one need only change each occurrence of "$w < 0$" to "$w \leq 0$".

The *No Case* rule is the same as the RELAX rule from standard shortest-path algorithms [4]. The *Lower Case* rule generates new constraints that guard against the possibility that a contingent link $\langle A, \ell, u, C\rangle$ might take on its minimum duration ℓ. Similarly, the *Upper Case* rule generates new constraints that guard against the possibility that a contingent link might take on its maximum duration u. However, unlike the *Lower Case* rule, which generates ordinary edges, the *Upper Case* rule generates upper-case edges that represent conditional wait constraints. For example, the generated edge $(X, [C\!:\!v - w], A)$ represents the conditional constraint that X must wait at least $(w - v)$ after A as long as C remains unexecuted. The *Cross Case* rule generates constraints that guard against one contingent link $\langle A, \ell, u, C\rangle$ taking on its minimum duration ℓ, while another contingent link $\langle A^K, \ell^K, u^K, K\rangle$ takes on its maximum duration u^K. Like the *Upper Case* rule, it generates upper-case edges. Finally, the *Label Removal* rule stipulates that in certain cases an upper-case edge has the force of an unconditional constraint. Each of the rules in Table 1 has been proven to be sound in the sense that if a valid and dynamic execution strategy σ satisfies the pre-existing edge(s) in the rule, then it must necessarily also satisfy the edge generated by that rule.

An important property of the rules in Table 1 is that they are *length-preserving* (i.e., the length of the generated edge is the same as the length of the corresponding path/edge from which it was derived). This property is exploited by the $O(N^5), O(N^4)$ and $O(N^3)$ DC-checking algorithms due to Morris and colleagues [13, 11, 12].

Each of the rules from Table 1 can be viewed as a path-transformation rule. For example, suppose that a path P contains two consecutive edges E_1 and E_2 to which one of the first four rules can be applied to generate a new edge E. If P' is the path obtained from P by replacing the two edges E_1 and E_2 by the new edge E, then we say that P has been transformed into P'. Similarly, the *Label Removal* rule can be viewed as a path-transformation rule that involves the replacement of just one edge.

Morris [11] provided a theoretical analysis of *semi-reducible* paths in STNU graphs that underlies much of the important work on DC-checking algorithms. A semi-reducible path is any path P that can be transformed into a path P' by any sequence of applications of the rules from Table 1 such that P' contains only ordinary or upper-case edges. Morris proved that an STNU is DC if and only if its graph has no semi-reducible negative loops (SRN loops). (Note that a negative loop containing only ordinary or upper-case edges can be further transformed by the *Upper Case* rule into a negative loop that contains only upper-case edges, which represents an inherently unsatisfiable cycle of constraints.) Central to his analysis was the process of "reducing away" lower-case edges – that is, performing a sequence of transformations that eliminate lower-case edges from the path.

Morris' $O(N^4)$ DC-checking algorithm searches for SRN loops by propagating forward from each contingent time-point C, looking for opportunities to reduce away the lower-case edge $(A, [c\!:\!\ell], C)$. (The path used to reduce away a lower-case edge e is called an *extension sub-path* for e. The path consisting of e followed by its extension sub-path is transformed into an ordinary or upper-case edge using the rules from Table 1.) To enable a modified use of Dijkstra's algorithm [4] to guide these forward propagations from each contingent time-point C, Morris' algorithm uses Bellman-Ford to compute (and update) a potential function for the OU-graph (i.e., the graph consisting of all ordinary and upper-case edges). He proved that in any non-DC network there must be an SRN loop in which the extension sub-paths used to reduce away lower-case edges are nested to a maximum depth of K, where K is the number of contingent links in the network. Thus, his algorithm performs at most K rounds, in each round doing forward propagations from each contingent time-point. After each round of edge generation, Bellman-Ford is run again, effectively recomputing

Table 2 The (non-length-preserving) *General Unordered Reduction* rule from Morris et al. [14].

Graphical Representation	Applicability Condition
$X \overset{[C:-y]}{\underset{-\ell}{\dashrightarrow}} A \xrightarrow{[c:\ell]} C$	$-y < -\ell$

the potential function to accommodate the newly generated edges. The overall complexity is thus $O(K((M + KN)N + N \log N)) = O(MNK + K^2N^2 + KN \log N)$ which, in dense graphs, reduces to $O(N^4)$. Hunsberger [9] subsequently introduced two improvements to the algorithm: (1) using a modified version of Dijkstra to compute a new potential function after each forward-propagation processing of a contingent time-point; and (2) using a heuristic to choose a “good” order in which to process the contingent time-points. These changes allowed newly generated edges to be inserted more quickly, and allowed the algorithm to terminate much earlier depending on the quality of the heuristic ordering. Empirically, the algorithm would often behave like an $O(N^3)$ algorithm, but in the worst case it was still $O(N^4)$.

More recently, Morris [12] presented a new approach to searching for lower-case reducing paths in an STNU graph: by propagating backward from negative edges. The intuition is that each negative edge could serve as the final edge (called a *moat edge*) in a path used to reduce away a lower-case edge. The advantages to this approach included that: (1) no potential function need be computed because the backward propagation could focus on propagating through only positive edges; and (2) the conflict between lower-case and upper-case edges from the same contingent link (which are not allowed in the *Cross Case* rule) effectively disappeared. As a result, at most N rounds of Dijkstra-like traversals are required, leading to a worst-case complexity of $O(N((M + N^2) + N \log N)) = O(N^3)$, which stands as the tightest upper bound on the complexity of the STNU-DC problem to date.

In contrast to all of the preceding algorithms, the algorithm presented in this paper, called the RUL$^-$ algorithm: (1) focuses on reducing away *upper-case* edges; (2) uses a *lower-bound* potential function to enable Dijkstra’s algorithm to guide the traversal of edges in the LO-graph (i.e., the graph consisting of all lower-case and ordinary edges); (4) only generates *ordinary* (i.e., *unlabeled*) edges; (5) includes a rule that is *not* length-preserving; and (6) only generates edges that terminate at either contingent or activation time-points. The RUL$^-$ algorithm uses one run of Bellman-Ford to initialize a potential function. It then does at most $2K$ rounds of constraint propagation. After each round, it uses a Dijkstra-like traversal to update the potential function for the next round. Thus, its overall complexity is $O(MN + K((M + KN) + N \log N)) = O(MN + K^2N + KN \log N)$. Although in the worst case this reduces to $O(N^3)$, in sparse graphs it can be much lower. For example, if $M = O(N \log N)$ and $K = O(\sqrt{N})$, it reduces to $O(N^2 \log N)$.

Notably, the one reduction from the earlier work that is not represented in Table 1 is the *General Unordered Reduction*, shown in Table 2, which is *not* length-preserving. It stipulates that, for a given contingent link $\langle A, \ell, u, C\rangle$, if X must wait at least y after A while C remains unexecuted, where $y > \ell$, then in *every* situation, X must wait at least ℓ after A, since C cannot execute before then. This rule will play an important role in this paper.

Table 3 The edge-generation rules for the RUL$^-$ algorithm.

Rule	**Graphical representation**	**Applicability Conditions**
Relax$^-$	$$\begin{tikzcd} P \arrow[r, "v"] \arrow[rr, dashed, bend right, "v+w"'] & Q \arrow[r, "w"] & R \end{tikzcd}$$	$Q \in \mathcal{T}_X, w < u^R - \ell^R, R \in \mathcal{T}_C$
Upper$^-$	$$\begin{tikzcd} P \arrow[r, "v"] \arrow[rr, dashed, bend right, "\max\{v-u, -\ell\}"'] & C \arrow[r, "{[C:-u]}"] & A \end{tikzcd}$$	(none)
Lower$^-$	$$\begin{tikzcd} A \arrow[r, "{[c:\ell]}"] \arrow[rr, dashed, bend right, "\ell+w"'] & C \arrow[r, "w"] & R \end{tikzcd}$$	$C \not\equiv R, w < u^R - \ell^R, R \in \mathcal{T}_C$

2.3 The RUL$^-$ Edge-Generation Rules

The RUL$^-$ algorithm uses three rules: Relax$^-$, Upper$^-$ and Lower$^-$.[3] As summarized in Table 3, each rule takes two consecutive edges (S, x, T) and (T, y, U) from the network and, if certain conditions are satisfied, generates a new *ordinary* edge from S to U. Although the rules have many similarities to rules from prior work [14, 13], they are distinguished in the following important ways.

- The RUL$^-$ rules only generate *ordinary* (i.e., *unlabeled*) edges; they *never* generate *lower-case* or *upper-case* edges.
- The Relax$^-$ and Lower$^-$ rules only generate edges terminating at contingent time-points.
- The value $u_R - \ell_R$, which represents the amount of uncertainty associated with the contingent link $\langle A_R, \ell_R, u_R, R\rangle$, plays an important role in the applicability conditions for the Relax$^-$ and Lower$^-$ rules.

The Relax$^-$ rule is identical to the Relax rule from the literature on shortest-path problems [4], except that it has very restrictive applicability conditions. The Relax$^-$ rule takes two ordinary edges (P, v, Q) and (Q, w, R), and generates the ordinary edge $(P, v{+}w, R)$. It only applies if Q is non-contingent, R is contingent, and $w < u_R - \ell_R$.

The Upper$^-$ rule is a combination of the *Upper Case* and *Label Removal* rules from Table 1, and the *General Unordered Reduction* rule from Table 2. It takes an ordinary edge (P, v, C), where C is contingent, and the original upper-case edge $(C, [C\!:\!-u], A)$ associated with C, and generates the ordinary edge (P, m, A), where $m = \max\{v - u, -\ell\}$. Note that the Upper$^-$ rule is *not* length-preserving.

The Lower$^-$ rule is similar to the *Lower Case* rule from Table 1; however, its applicability conditions are quite different. The Lower$^-$ rule takes a lower-case edge $(A, [c\!:\!\ell], C)$ and an ordinary edge (C, w, R), and generates the ordinary edge $(A, \ell + w, R)$. Unlike the *Lower Case* rule, the Lower$^-$ rule only applies if R is a contingent time-point. For this reason, it can be applied whenever $w < u^R - \ell^R$, whereas the *Lower Case* rule only applies if $w < 0$.

▶ **Proposition 1.** *The* Relax$^-$ *rule from Table 3 is sound.*

Proof. The Relax$^-$ rule is identical to the sound *No Case* rule from Table 1, except that it has more restrictive applicability conditions. Thus, it too must be sound. ◀

[3] The RUL$^-$ rules have the same form as the RUL$^+$ rules developed in prior work [1], but their applicability conditions are quite different. As a result, the RUL$^+$ rules do not form the basis of an efficient DC-checking algorithm, whereas the RUL$^-$ rules do. To avoid confusion, the RUL$^+$ rules are not discussed further in this paper.

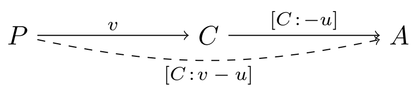

Figure 2 An upper-case edge generated during the proof of Prop. 2.

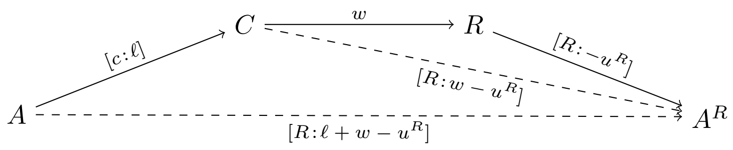

Figure 3 Upper-case edges generated during the proof of Prop. 3.

▶ **Proposition 2.** *The* Upper$^-$ *rule from Table 3 is sound.*

Proof. The soundness of the Upper$^-$ rule is obtained from the soundness of the *Upper Case* and *Label Removal* rules from Table 1, and the *General Unordered Reduction* rule from Table 2. First, let (P, v, C) and $(C, [C\!:\!-u], A)$ be the pre-existing edges in the Upper$^-$ rule, as shown in Table 3. The *Upper Case* rule generates the *upper-case* edge $(P, [C\!:\!v-u], A)$, as illustrated in Fig. 2.

Case 1: $v - u \geq -\ell$ (i.e., $m = v - u$). In this case, the *Label Removal* rule applies, yielding the *unlabeled* edge $(P, v-u, A)$ (i.e., (P, m, A)).

Case 2: $v - u < -\ell$ (i.e., $m = -\ell$). In this case, σ satisfying the *upper-case* edge $(P, [C\!:\! v-u], A)$ implies that in each situation ω, either P waits at least $(u - v)$ after A or P executes at or after C.[4] Therefore, either $[\sigma(\omega)]_P \geq [\sigma(\omega)]_A + (u-v) > [\sigma(\omega)]_A + \ell$ or $[\sigma(\omega)]_P > [\sigma(\omega)]_C \geq [\sigma(\omega)]_A + \ell$. As a result, in each situation ω, σ necessarily satisfies $[\sigma(\omega)]_P \geq [\sigma(\omega)]_A + \ell = [\sigma(\omega)]_A - m$, represented by the edge (P, m, A). ◀

▶ **Proposition 3.** *The* Lower$^-$ *rule from Table 3 is sound.*

Proof. The soundness of the Lower$^-$ rule derives from the soundness of several of the rules from Table 1, as follows. Let $(A, [c\!:\!\ell], C)$ and (C, w, R) be the two pre-existing edges for the Lower$^-$ rule, as shown in Table 3, where $w < u^R - \ell^R$ and R is contingent.

Case 1: $w < 0$. Here, the *Lower Case* rule applies, yielding the desired edge.

Case 2: $w \in [0, u^R - \ell^R)$. First, the *Upper Case* rule applied to the edges (C, w, R) and $(R, [R\!:\!-u^R], A^R)$ yields the *upper-case* edge $(C, [R\!:\!w - u^R], A^R)$, as illustrated in Fig. 3. And since $w - u^R < -\ell^R < 0$, the *Cross Case* rule can then be applied to yield the *upper-case* edge $(A, [R\!:\!\ell + w - u^R], A^R)$, also shown in Fig. 3. Since σ is valid, it must satisfy this generated *upper-case* edge. Therefore, in each situation ω, either A executes at least $(-\ell - w + u^R)$ after A^R; or A executes at or after R. Hence, one of the following holds:

(1) $[\sigma(\omega)]_A \geq [\sigma(\omega)]_{A^R} - \ell - w + u^R$; or

(2) $[\sigma(\omega)]_A \geq [\sigma(\omega)]_R$.

For (1), it follows that $[\sigma(\omega)]_R - [\sigma(\omega)]_A \leq [\sigma(\omega)]_R - [\sigma(\omega)]_{A^R} + \ell + w - u^R$. And, since the semantics of contingent links ensures that $[\sigma(\omega)]_R - [\sigma(\omega)]_{A^R} \leq u^R$, it follows that $[\sigma(\omega)]_R - [\sigma(\omega)]_{A^R} + \ell + w - u^R \leq u^R + \ell + w - u^R = \ell + w$. For (2), $[\sigma(\omega)]_R - [\sigma(\omega)]_A \leq 0 < \ell + w$, since $\ell > 0$ and $w \geq 0$. Thus, in either case, σ satisfies the edge $(A, \ell + w, R)$. ◀

[4] The semantics of satisfying a generated upper-case edge is detailed by Hunsberger [8].

▶ **Theorem 1** (Completeness of the RUL$^-$ rules)**.** *Let $\mathcal{S}$ be any STNU; let $\mathcal{G}$ be the graph for $\mathcal{S}$ (prior to any application of the* RUL$^-$ *rules); and let $\mathcal{G}^*$ be the closure of $\mathcal{G}$ under the* RUL$^-$ *rules. If the LO-graph for $\mathcal{G}^*$ has no negative loops (i.e., is consistent when viewed as an STN, ignoring the alphabetic labels on its lower-case edges), then $\mathcal{S}$ must be DC.*

Proof. Let $\mathcal{S}$ be any STNU with graph $\mathcal{G}$ prior to any application of the RUL$^-$ rules. Suppose that $\mathcal{S}$ is not DC. By Morris [11], $\mathcal{G}$ must contain a semi-reducible negative loop π that is *breach-free* (i.e., if $(A, [c\!:\!\ell], C)$ is a lower-case edge in π, and $\mathcal{P}$ is the *extension sub-path* in π that is used to "reduce away" that lower-case edge, then $\mathcal{P}$ does not contain any occurrence of the corresponding upper-case edge $(C, [C\!:\!-u], A)$). By a similar argument, it can be shown that no loss of generality results from assuming that the extension sub-path of any lower-case edge $(A, [c\!:\!\ell], C)$ in π does not include any occurrence of the contingent time-point C.

First note that if π contains no upper-case edges, then π is in the LO-graph, contradicting that the LO-graph has no negative loops. Thus, π must have one or more upper-case edges.

In what follows, let Upper† denote the restriction of the Upper$^-$ rule to the case where $v \geq u - \ell$ (i.e., the *length-preserving* case). And let RUL† denote the set of rules {Relax$^-$, Lower$^-$, Upper† }. Note that the RUL† rules are length-preserving. The following shall focus on the use of the RUL† rules to reduce away all of the upper-case edges in (a suitably transformed) π. However, in certain exceptional cases, the non-length-preserving case of the Upper$^-$ rule will be applied.

⇛ This proof is supported by Morris' *analysis* of semi-reducible paths in which each lower-case edge is followed by an *extension sub-path* that can be used to reduce it away by applying the rules from Table 1. In contrast, the RUL$^-$ algorithm uses the rules from Table 3 to reduce away *upper-case edges.* The proof uses the properties of semi-reducible paths to confirm that applying the RUL$^-$ algorithm to the semi-reducible path π necessarily leads to a "Non-DC" conclusion.

Suppose that E is some upper-case edge $(C, [C\!:\!-u], A)$ in π that cannot be reduced away by the RUL† rules. Consider back-propagating from C using the Relax$^-$ and Lower$^-$ rules. If this process ever results in an edge (T, s, C), where $s \geq u - \ell$, then that edge could be used to reduce away E via the Upper† rule, since $s - u \geq -\ell$. To prevent this, one of the following must happen:

(1) The lower-case edge $(A, [c\!:\!\ell], C)$ for the *same* contingent link is encountered, resulting in a path of the form, $(A, [c\!:\!\ell], C, \ldots, C, [C\!:\!-u], A)$. Now, given that the original path π was breach-free, the sub-path used to reduce away the lower-case edge $(A, [c\!:\!\ell], C)$ must be a proper prefix of the path from C to C; and it must have negative length. Thus, the path must have the form $(A, [c\!:\!\ell], C, \ldots, X, w, C, [C\!:\!-u], A)$, where the path from C to X has some length $\delta < 0$, and where $w < u - \ell$, since back-propagation continued from C back to X and beyond. In this case, since $w - u < -\ell$, the non-length-preserving case of the Upper$^-$ rule generates the edge $(X, -\ell, A)$. But then the loop from A to A has length $\ell + \delta - \ell = \delta < 0$. Since this loop consists only of ordinary and lower-case edges, it contradicts the hypothesis that the LO-graph has no negative loops.

(2) A two-edge path (X, r, P, v, C) is encountered, where P is contingent, but (X, r, P) is an ordinary edge, and $v < u - \ell$. (This blocks the Relax$^-$ rule since P is contingent, and blocks the Lower$^-$ rule since (X, r, P) is ordinary.) In this case, insert the path $(P, [P\!:\!-u^P], A^p, [p\!:\!\ell^p], P)$. Then use the Lower$^-$ rule to generate the edge $(A^p, \ell^P + v, C)$, as shown in Fig. 4. Now, if $\ell^P + v \geq u - \ell$, then E can be reduced away. If not, then further back-propagation is blocked by the upper-case edge $(P, [P\!:\!-u^P], A^P)$.

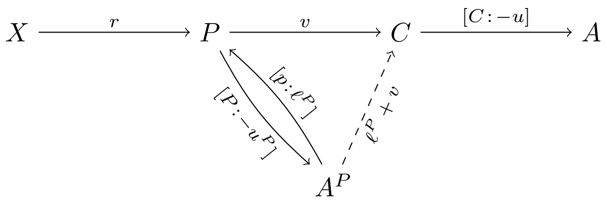

Figure 4 Inserting labeled edges into the path π in Case 2 of the proof of Theorem 1.

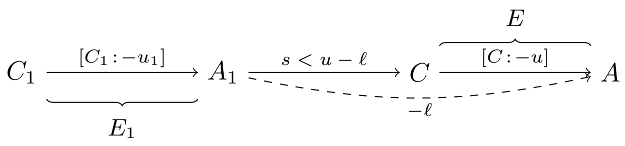

Figure 5 An upper-case edge E_1 blocking the reducing-away of another upper-case edge E.

In view of the above analysis, the only way that the back-propagation could fail to reduce away the upper-case edge E in π is if it is blocked by some upper-case edge E_1, as illustrated in Fig. 5. The upper-case edge E_1 could be one that was originally in π, or one that was added in Case (2) above, where $\ell^p + v < u - \ell$. In either case, note that the non-length-preserving case of the Upper$^-$ rule can be used to generate a negative edge from E_1's activation time-point A_1 to E's activation time-point A, as illustrated in the figure. Thus, if the recursive processing of successive upper-case edges ever encounters a repeat, it would signal a negative loop in the LO-graph, contradicting the hypothesis.

As a result, the recursive processing of some upper-case edge E_i must *not* be blocked (i.e., E_i can be reduced away) which implies that the processing of the preceding upper-case edge E_{i-1} can resume. Continuing in this way, either E_i or one of its successor upper-case edges in π must be unblocked (i.e., can be reduced away), and so on, until all upper-case edges have been reduced away from π. Since the reducing away of upper-case edges only uses the length-preserving Upper† rule, the transformed π (now in the LO-graph) still has negative length and, thus, contradicts the hypothesis that the LO-graph is consistent. (The non-length-preserving case of the Upper$^-$ rule was used only to signal inconsistencies that terminated the recursion; it was not used in cases where the recursion continued.)

Finally, note that when inserting the upper-case edge $(P, [P:-u^P], A^P)$ in Case (2) above, it was assumed that the process of reducing away that upper-case edge could not affect the semi-reducibility of π. That is true, but it must be proven, as follows. First, since P was already in π, and P cannot appear in the extension sub-path $\mathcal{P}_P$ used to reduce away the lower-case edge $(A^P, [p:\ell^P], P)$, it follows that introducing the upper-case edge for P into π cannot cause a breach for the lower-case edge for P. Next, suppose that introducing the upper-case edge for P, and subsequently reducing it away, consumed the *moat* edge in the extension sub-path $\mathcal{P}_Q$ used to reduce away the lower-case edge $(A^Q, [q:\ell^Q], Q)$, as illustrated in Fig. 6. Since all proper prefixes of the extension sub-path $\mathcal{P}_Q$ must have non-negative length (otherwise they could be used to reduce away the lower-case edge), $a \geq 0$ and $a+b \geq 0$. However, the entire extension sub-path must have negative length; thus, $a + b + c < 0$, which implies that $c < -a - b \leq 0$. Now, $c < 0$ implies that $c + v < v < u^P - \ell^P$; hence the path used to reduce away the upper-case edge extends backward, beyond X, to some W. Thus, $b + c + v \geq u^P - \ell^P$ (to enable the use of the Upper† rule). But then:

$$u^P - \ell^P \leq b + c + v < b + (-a - b) + v = -a + v \leq v < u^P - \ell^P$$

which is a contradiction. Thus, our assumption that inserting the upper-case edge involving P caused the path π to become non-semi-reducible was wrong. ◀

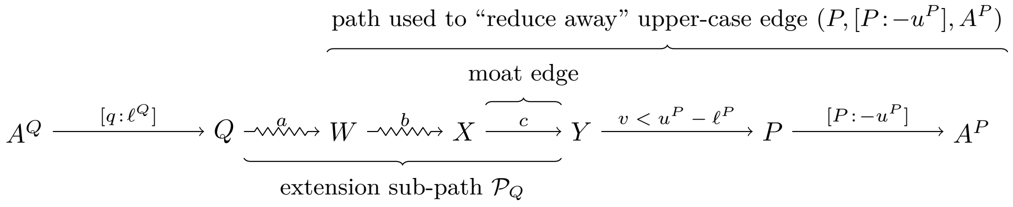
path used to "reduce away" upper-case edge $(P, [P\!:\!-u^P], A^P)$
moat edge
A^Q
$[q\!:\!\ell^Q]$
Q
a
W
b
X
c
Y
$v < u^P - \ell^P$
P
$[P\!:\!-u^P]$
A^P
extension sub-path $\mathcal{P}_Q$

Figure 6 A path discussed in the proof of Theorem 1, where $a+b+c<0$ and $b+c+v \geq u^P - \ell^P$.

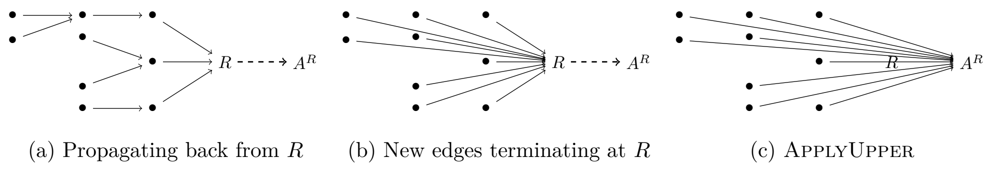
R
A^R
R
A^R
R
A^R
(a) Propagating back from R
(b) New edges terminating at R
(c) ApplyUpper

Figure 7 Reducing away an upper-case edge $(R, [R\!:\!-u^R], A^R)$ with the RUL$^-$ algorithm.

3 The RUL$^-$ DC-checking Algorithm

This section presents the main contribution of the paper: the RUL$^-$ DC-checking algorithm for STNUs. When applied to a given semi-reducible negative loop, the algorithm essentially follows the structure of the proof of Theorem 1. Of course, it is not generally given a semi-reducible negative loop in advance; therefore, it effectively carries out its back-propagation along all potential paths in parallel.

The pseudo-code for the RUL$^-$ DC-checking algorithm is given in Algorithm 1, where it is called DC-Check. The constraint-propagation functions, CloseRelaxLower, ApplyRelaxLower and ApplyUpper, that are used by DC-Check are collected in Algorithm 2. And the InitPotential, UpdatePotential and NegativeCycle functions, used to initialize, update, and verify the consistency of a potential function h for the LO-graph $\mathcal{E}^o \cup \mathcal{E}^\ell$ are collected in Algorithm 3.

The input to the DC-Check function is an STNU graph $\mathcal{G}$. The algorithm focuses on reducing away each upper-case edge $(R, [R\!:\!-u^R], A^R)$, as illustrated in Fig. 7. First, it uses the CloseRelaxLower and ApplyRelaxLower functions to propagate backward from the contingent time-point R along edges in the LO-graph $\mathcal{E}^o \cup \mathcal{E}^\ell$ (Fig. 7a) and then generate new edges terminating at R using the Relax$^-$ and Lower$^-$ rules (Fig. 7b). Then for every edge terminating at R, the ApplyUpper function uses the Upper$^-$ rule to generate new edges terminating at the activation time-point A^R (Fig. 7c), effectively providing all ways of reducing away the upper-case edge $(R, [R\!:\!-u^R], A^R)$.[5]

If the backward propagation from R is ever blocked by another upper-case edge E_1, the algorithm suspends processing of R and recursively begins to process E_1. Similarly, the processing of E_1 could subsequently be blocked by other upper-case edges, which would require suspending the processing of E_1, and so on. At each stage, the processing of an upper-case edge is only resumed when *all* of the blocking upper-case edges have been fully processed. As in the proof of Theorem 1, if the algorithm ever recursively encounters the same upper-case edge twice, it immediately terminates with a *false* (i.e., "Non-DC") answer.

[5] The algorithm could postpone using the non-length-preserving case of the Upper$^-$ rule until all other propagations were done, but it is more convenient to apply both cases of RUL$^-$ when the opportunity arises.

Algorithm 1: DC-Check, the RUL$^-$ DC-checking algorithm.

```
Input: $\mathcal{G}$                                              ▷ An STNU graph
Output: $\langle True, \mathcal{G}\rangle$, if DC; $False$, otherwise        ▷ Side Effect: Modifies $\mathcal{G}$
1  $h \longleftarrow$ InitPotential ($\mathcal{G}$)
2  if NegativeCycle ($\mathcal{G}, h$) then return $False$
3  $\mathcal{R} \longleftarrow \mathcal{T}_C$
4  $S \longleftarrow$ new empty stack
5  push $Z$ onto $S$
6  while $S$ is not empty do
7      $R \longleftarrow$ top of $S$                               ▷ Do not pop from stack
8      $\mathcal{G} \longleftarrow$ CloseRelaxLower ($\mathcal{G}, h, R$)
9      $\mathcal{G} \longleftarrow$ ApplyUpper ($\mathcal{G}, R$)
10     $h \longleftarrow$ UpdatePotential ($\mathcal{G}, h, A^R$)
11     if NegativeCycle ($\mathcal{G}, h$) then return $False$
12     if $\exists R' \in \mathcal{R}$ such that $w_{A^{R'}R} < u^R - \ell^R$ then
13         if $R' \in S$ then return $False$
14         else push $R'$ onto $S$
15     else
16         $\mathcal{R} \longleftarrow \mathcal{R} \setminus \{R\}$
17         pop $R$ from top of $S$
18         if $\mathcal{R}$ is non-empty and $S$ is empty then
19             push top element of $\mathcal{R}$ onto $S$          ▷ But keep it in $\mathcal{R}$
20     return $\langle True, \mathcal{G}\rangle$
```

But if it successfully completes the processing of all K upper-case edges without encountering a negative loop in the LO-graph, it terminates with a *true* (i.e., “DC”) answer.

To efficiently perform the back-propgation processes, the algorithm uses a potential function h for the LO-graph. (The LO-graph can be viewed as an STN by ignoring any alphabetic labels on its lower-case edges.) The InitPotential function uses the Single *Sink* version of Bellman-Ford to create a potential function that specifies a lower bound for each time-point in the network. In particular, if $d(X)$ is the distance from X to some arbitrary sink node, then $h(X) = -d(X)$ specifies a lower bound for X. If the LO-graph has no negative cycles, then the potential function provides a solution to the LO-graph, as an STN. The potential function h is then used, as in Johnson's algorithm [4], to transform the weight δ of each edge (X, δ, Y) into a non-negative value, $h(X) + \delta - h(Y) = -d(X) + \delta + d(Y) \geq 0$, thereby enabling Dijkstra's algorithm to be used to guide the back-propagation from each contingent time-point R in the CloseRelaxLower function. Afterward, the ApplyUpper function is used to generate new edges terminating at the activation time-point A^R. Since these edges are derived from an upper-case edge, but the potential function is based on the LO-graph, the potential function must be updated to accommodate those new edges. The UpdatePotential function handles this chore using a modified version of Dijkstra that allows negative edges as long as they all have the same destination – in this case, they all terminate at A^R.

After updating the potential function, the main algorithm checks whether there is an upper-case edge $(C', [C'{:}-u'], A')$ that has not yet been processed for which there is an edge (A', s, R) for which $s < u^R - \ell^R$, which would block further back-propagation from R. (It may help to recall Fig. 5, substituting R for C, and $s < u^R - \ell^R$ for $v < u - \ell$.) If so, it recursively processes that upper-case edge. Once that process completes, it checks

Algorithm 2: Constraint-propagation functions for DC-CHECK algorithm.

```
1  function CloseRelaxLower ($\mathcal{G}$, an STNU graph; $h$, a potential function; $R \in \mathcal{T}_C$):
       Pre: $h(P) \geq h(Q) - w$ for each $(P, w, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell$
       Post: All new edges $(P, v, R)$ that can be generated by Relax$^-$ and Lower$^-$
             have been added to $\mathcal{E}^o$
2      $\mathcal{Q} \longleftarrow$ new priority queue
3      foreach $(Q, x, R) \in \mathcal{E}^o \cup \mathcal{E}^\ell$ do  insert $Q$ into $\mathcal{Q}$ with priority $h(Q) + x$
4      while $\mathcal{Q}$ is not empty do
5          $Q \longleftarrow$ pop min priority node from $\mathcal{Q}$
6          foreach $(P, v, R) \in$ ApplyRelaxLower $(\mathcal{G}, Q, R)$ do
7              if $v < w_{PR}$ then insert edge $(P, v, R)$ into $\mathcal{E}^o$
8              $w_{PR} \longleftarrow \min\{w_{PR}, v\}$
9              if $P \in \mathcal{Q}$ then decrease priority of $P$ to $h(P) + w_{PR}$
10             else insert $P$ into $\mathcal{Q}$ with priority $h(P) + w_{PR}$
11     return $\mathcal{G}$
12 function ApplyRelaxLower ($\mathcal{G}$, an STNU graph; $Q \in \mathcal{T}$; $R \in \mathcal{T}_C$):
       Output: The set of all edges $(P, w, R)$ obtained by applying Relax$^-$ or
               Lower$^-$ to the path $(P, w_{PQ}, Q, w_{QR}, R)$ in $\mathcal{E}^o \cup \mathcal{E}^\ell$
13     if $w_{QR} \geq u^R - \ell^R$ then return $\emptyset$          ▷ Relax$^-$ and Lower$^-$ do not apply
14     else if $Q \in \mathcal{T}_C$ then  return $(A^Q, \ell^Q + w_{QR}, R)$          ▷ Apply
        Lower$^-$
15     else return $\{(P, w_{PQ} + w_{QR}, R)\}_{(P, w_{PQ}, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell, P \in \mathcal{T} \setminus \{R\}}$          ▷ Apply
        Relax$^-$
16 function ApplyUpper ($\mathcal{G}$, an STNU graph; $R \in \mathcal{T}_C$):
       Output: All new edges $(P, w, A^R)$ obtained by applying Upper$^-$ to paths
               $(P, v, R, [R{:}{-u^R}], A^R)$ in $\mathcal{E}^o \cup \mathcal{E}^\ell$ have been added to $\mathcal{E}^o$
17     foreach $(P, v, R) \in \mathcal{E}^o$ do
18         if $v < u^R - \ell^R$ then $w_{PA^R} \longleftarrow \min\{w_{PA^R}, -\ell^R\}$
19         else $w_{PA^R} \longleftarrow \min\{w_{PA^R}, v - u^R\}$
20         insert edge $(P, w_{PA^R}, A^R)$ into $\mathcal{E}^o$
21     return $\mathcal{G}$
```

whether there are any other as-yet-unprocessed upper-case edges that meet that criterion. If so, it processes each one of them in turn. Once all such upper-case edges are successfully processed, the algorithm processes R again, from scratch, and then moves to process any other as-yet-unprocessed upper-case edges (even though they may not be blocking for R) until all such edges have been processed. By maintaining separate stacks, $\mathcal{R}$ and S, respectively, of contingent time-points that have not yet been processed, and those whose processing is in progress, the algorithm is guaranteed to perform at most $2K$ rounds. (Each round terminates by pushing a time-point $R' \in \mathcal{R}$ onto S, or removing a time-point R from both $\mathcal{R}$ and $\mathcal{S}$. Since no time-point is ever pushed onto S or popped off of S more than once, it follows that at most $2K$ rounds can be performed.) If updating the potential function ever fails, then the algorithm immediately returns *false* (i.e., "Non-DC"). If the algorithm ever recursively encounters the same upper-case edge twice, it immediately returns "Non-DC". If none of those things happen then, after at most $2K$ rounds, the algorithm returns *true* (i.e., "DC").

Algorithm 3: Functions to initialize/update/verify lower-bound potential function.

```
1  function InitPotential ($\mathcal{G}$, an STNU graph):
       Post: $h(P) \geq h(Q) - w$ for each $(P, w, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell$, unless graph has neg. cycle
       ▷ $N - 1$ rounds of Bellman-Ford, where $N = |\mathcal{T}|$
2      foreach $P \in \mathcal{T}$ do $h(P) \longleftarrow 0$
3      for $i = 1$ to $N - 1$ do
4          foreach edge $(P, w, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell$ do
5              $h(P) \longleftarrow \max\{h(P), h(Q) - w\}$        ▷ (i.e.,
                 $d(P) \longleftarrow \min\{d(P), d(Q) + w\}$)
6      return $h$
7  function NegativeCycle ($\mathcal{G}$, an STNU graph; $h$, a potential function for $\mathcal{G}$):
       Output: True if $\mathcal{E}^o \cup \mathcal{E}^\ell$ has a negative cycle; False otherwise
       ▷ Last round of Bellman-Ford
8      foreach edge $(P, w, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell$ do
9          if $h(P) < h(Q) - w$ then return True
10     return False
11 function UpdatePotential ($\mathcal{G}$, STNU graph; $h$, potential function for $\mathcal{G}$; $A \in \mathcal{T}$):
       Pre: $h(P) \geq h(Q) - w$ for each $(P, w, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell$ where $Q \neq A$
       Post: $h'(P) \geq h'(Q) - w$ for each $(P, w, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell$ unless graph has a neg.
             cycle
12     $h' \longleftarrow h$
13     $\mathcal{Q} \longleftarrow$ new priority queue
       ▷ Priority of each time-point = amount its lower-bound changes
14     insert $A$ into $\mathcal{Q}$ with priority 0
15     while $\mathcal{Q}$ is not empty do
16         $Q \longleftarrow$ extract min. priority node from $\mathcal{Q}$
17         foreach $(P, w, Q) \in \mathcal{E}^o \cup \mathcal{E}^\ell$ do
18             if $h'(P) < h'(Q) - w$ (i.e., $d(P) > d(Q) + w$) then
19                 $h'(P) \longleftarrow h'(Q) - w$ (i.e., $d(P) \longleftarrow d(Q) + w$)
20                 if $P \in \mathcal{Q}$ then decrease priority of $P$ in $\mathcal{Q}$ to $h(P) - h'(P)$
21                 else insert $P$ into $\mathcal{Q}$ with priority $h(P) - h'(P)$
22     return $h'$
```

▶ **Corollary 1.** *The* RUL$^-$ *algorithm is sound and complete for the STNU-DC problem.*

Proof. The RUL$^-$ rules are sound by Propositions 1–3. And if an STNU graph $\mathcal{G}$ has a semi-reducible negative loop, then by the proof of Theorem 1, the RUL$^-$ algorithm will return "Non-DC". ◀

4 Conclusion

This paper introduced a new algorithm, called the RUL$^-$ algorithm, for solving the DC-checking problem for STNUs. The algorithm, which is proven to be sound and complete, uses three constraint-propagation rules, Relax$^-$, Lower$^-$ and Upper$^-$, that differ from other approaches in that they generate only ordinary edges, and they focus on generating edges that terminate either in contingent or activation time-points. As a result, the algorithm performs at most $2K$ rounds. Since each round generates at most N new edges, the algorithm generates at most $2KN$ new edges overall. The constraint propagation in each round is guided by Dijkstra's algorithm, using a lower-bound potential function (as in Johnson's algorithm). Each round of edge generation is interleaved with another run of Dijkstra to update the lower-bound potential function to accommodate the new edges. Thus, the complexity of each round is $O((M + KN) + N \log N)$; and the overall complexity of the RUL$^-$ algorithm is $O(MN + K^2N + KN \log N)$, where the $O(MN)$ term arises from the use of Bellman-Ford to initialize the potential function. For sparse networks, this upper bound on the complexity of the STNU-DC problem is tighter than the previous best-known bound of $O(N^3)$. Future work will focus on experimentally evaluating the RUL$^-$ algorithm and Morris' $O(N^3)$ algorithm to see whether further enhancements might be found.

References

1 Massimo Cairo and Romeo Rizzi. Dynamic Controllability Made Simple. In Sven Schewe, Thomas Schneider, and Jef Wijsen, editors, *24th International Symposium on Temporal Representation and Reasoning (TIME 2017)*, volume 90 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:16, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TIME.2017.8`.

2 Carlo Combi, Luke Hunsberger, and Roberto Posenato. An algorithm for checking the dynamic controllability of a conditional simple temporal network with uncertainty. In *ICAART 2013*, volume 2, pages 144–156, 2013. `doi:10.5220/0004256101440156`.

3 Patrick R. Conrad and Brian C. Williams. Drake: An efficient executive for temporal plans with choice. *Journal of Artificial Intelligence Research (JAIR)*, 42:607–659, 2011. `doi:10.1613/jair.3478`.

4 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2001.

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

6 Robert Effinger, Brian Williams, Gerard Kelly, and Michael Sheehy. Dynamic controllability of temporally-flexible reactive programs. In *19th International Conference on Automated Planning and Scheduling (ICAPS-2009)*, 2009.

7 Luke Hunsberger. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *TIME 2009*, pages 155–162, 2009. `doi:10.1109/TIME.2009.25`.

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

9 Luke Hunsberger. New techniques for checking dynamic controllability of simple temporal networks with uncertainty. In *Lecture Notes in Computer Science*, volume 8946, pages 170–193. Springer, 2015.

10 Lina Khatib, Paul H. Morris, Robert A. Morris, and Francesca Rossi. Temporal constraint reasoning with preferences. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 322–327, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers, Inc.

11 Paul Morris. A structural characterization of temporal dynamic controllability. In *CP 2006*, volume 4204, pages 375–389, 2006. `doi:10.1007/11889205_28`.

12 Paul Morris. Dynamic controllability and dispatchability relationships. In *Integration of AI and OR Techniques in Constraint Programming*, volume 8451 of *Lecture Notes in Computer Science*, pages 464–479. Springer, 2014.

13 Paul H. Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *AAAI-05/IAAI-05*, pages 1193–1198, 2005.

14 Paul H. Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *IJCAI 2001*, pages 494–502, 2001.

15 Francesca Rossi, Kristen Brent Venable, and Neil Yorke-Smith. Uncertainty in soft temporal constraint problems: A general framework and controllability algorithms for the fuzzy case. *Journal of Artificial Intelligence Research*, 27:617–674, 2006.