Dipartimento di Informatica
Università degli Studi di Verona

Rapporto di ricerca
Research report

RR 109/2021

A note on speeding up DC-checking for STNUs

Luke Hunsberger
hunsberger@vassar.edu

Roberto Posenato
roberto.posenato@univr.it

Dipartimento di Informatica
Università degli Studi di Verona

Rapporto di ricerca
Research report
July 2021

RR 109/2021

A note on speeding up DC-checking for STNUs

Luke Hunsberger
hunsberger@vassar.edu

Roberto Posenato
roberto.posenato@univr.it

Questo rapporto è disponibile su Web all'indirizzo:
This report is available on the web at the address:
http://hdl.handle.net/11562/1045707

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. 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 (equivalently, the number of edges in the STNU graph), 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 implementation of the algorithm that improves its performance in practice by an order of magnitude, as demonstrated by a thorough empirical evaluation.

Algorithm Research Group

Contents

1 Background **1**
- 1.1 Simple Temporal Networks with Uncertainty 5
- 1.2 Dynamic Controllability of an STNU 7

2 Existing DC-Checking Algorithms for STNUs **9**
- 2.1 Morris' 2006 $O(n^4)$ DC-Checking Algorithm 10
- 2.2 Morris' 2014 DC-Checking Algorithm 16
- 2.3 The RUL$^-$ DC-Checking Algorithm 20

3 A New Approach to the RUL$^-$ Algorithm **26**
- 3.1 Pseudocode for the New RUL DC-Checking Algorithm 32

4 Experimental Evaluation **39**

References **55**

1 Background

Simple Temporal Networks (STNs) were introduced by Dechter et al. (1991) to facilitate reasoning about actions subject to temporal constraints. An STN is a data structure that contains real-valued variables called *time-points* and binary difference constraints on those time-points.

Definition 1 (STN)**.** A Simple Temporal Network (STN) is a pair $(\mathcal{T}, \mathcal{C})$ where:

- $\mathcal{T} = \{X_0, X_1, X_2, \ldots, X_{n-1}\}$ is a set of real-valued variables called *time-points*; and
- $\mathcal{C}$ is a set of binary difference constraints (also called *ordinary* constraints) over time-points in $\mathcal{T}$. In particular, each constraint in $\mathcal{C}$ has the form $Y - X \leq \delta$ for some $X, Y \in \mathcal{T}$, and some $\delta \in \mathbb{R}$.

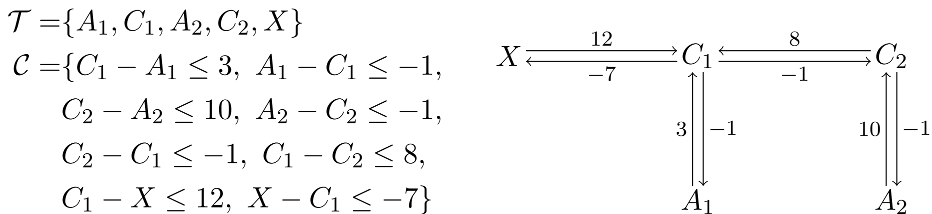


Figure 1: A sample STN (left) and its corresponding graph (right)

Algorithm 1: The single-source version of the Bellman-Ford algorithm

```
1 function BellmanFord (𝒢):
    Input: 𝒢 = (𝒯, ℰ), an STN graph
    Output: d, a potential fn. such that d(Y) − d(X) ≤ δ for each edge
            (X, δ, Y) ∈ 𝒢, unless 𝒢 has negative cycle
2   foreach X ∈ 𝒯 do d(X) := 0
3   for i = 1 to |𝒯| − 1 do
4       foreach edge (X, δ, Y) ∈ 𝒢 do
5           d(Y) := min{d(Y), d(X) + δ}        // Ensures that d(Y) − d(X) ≤ δ
    // Checks for a negative cycle
6   foreach edge (X, δ, Y) ∈ 𝒢 do
7       if d(Y) − d(X) < δ then return ⊥        // Found a negative cycle
8   return d                                     // Found a solution to the STN
```

The left side of Figure 1 gives a sample STN with five time-points and eight constraints. (The time-point names are chosen to facilitate comparisons with later networks.) It is common to let n represent the number of time-points in an STN, and m the number of constraints. In addition, in this paper, for convenience, we may sometimes identify the indices $0, 1, \ldots, n-1$ with the corresponding time-points $X_0, X_1, \ldots, X_{n-1}$.

Definition 2 (STN graph)**.** Each STN $\mathcal{S} = (\mathcal{T}, \mathcal{C})$ has a corresponding graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$ where the time-points in $\mathcal{T}$ serve as the nodes in the graph, and the constraints in $\mathcal{C}$ correspond to edges in the graph. In particular, for each ordinary constraint $(Y - X \leq \delta)$ in $\mathcal{C}$, there is a labeled directed edge, called an *ordinary* edge, $X \xrightarrow{\delta} Y$ in $\mathcal{E}$.

The righthand side of Figure 1 illustrates the graph for the sample STN from the lefthand side of the figure.

The Fundamental Theorem of STNs states that an STN is consistent (i.e., has a solution as a constraint satisfaction problem) if and only if its graph has no negative cycles (i.e., no negative-length loops) (Dechter et al., 1991; Hunsberger, 2014b).

The Bellman-Ford SSSP algorithm. The consistency of an STN can be determined, for example, by the Bellman-Ford Single-Source Shortest-Paths (SSSP) algorithm (Cormen, Leiserson, Rivest, & Stein, 2009). Algorithms in this paper implement Bellman-Ford as either a single-*source* or single-*sink,* shortest-paths algorithm in which the source/sink node is a new node $S \notin \mathcal{T}$. For consistent networks, the single-source version of the algorithm, shown in Algorithm 1, 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

Algorithm 2: A typical implementation of Dijkstra's SSSP algorithm

```
1  function Dijkstra (𝒢, S):
       Input: 𝒢 = (𝒯, ℰ), an STN graph with non-negative edges; S ∈ 𝒯
       Output: d, a fn. specifying the min. distance from source node S to
               each node in 𝒢
2      foreach X ∈ 𝒯 do d(X) := ∞ // d(X) = distance from source node S to X
3      d(S) := 0
4      𝒬 := a new priority queue
5      𝒬.insert(S, 0)
6      while (¬𝒬.empty()) do
7          V := 𝒬.extractMinNode()
8          d(V) := 𝒬.key(V)
9          foreach (V, δ, W) ∈ ℰ do
               𝒬.insertOrDecreaseKeyIfSmaller(W, d(V) + δ)
10     return d
```

Operator	Output/Side Effect
$\mathcal{Q}.state(x)$	returns the *state* of x, one of `notYetInQ`, `inQ` or `alreadyPopped`
$\mathcal{Q}.key(x)$	returns the key/priority of x (which must be in the queue)
$\mathcal{Q}.empty()$	returns *true* if the queue is currently empty
$\mathcal{Q}.extractMinNode()$	pop the datum with minimum priority off the queue
$\mathcal{Q}.insert(x, p)$	insert datum x into the queue $\mathcal{Q}$ with priority/key p
$\mathcal{Q}.decreaseKey(x, p)$	decrease the priority/key of x (which must be in the queue) to p
$\mathcal{Q}.clear()$	clear the contents of the queue

Table 1: Operators provided by a priority queue, $\mathcal{Q}$

to X. To ensure connectedness, $d(X)$ is initialized to 0 for each $X \in \mathcal{T}$ (Line 2), simulating the presence of an edge from S to X of length 0. The algorithm then updates the distance function in $O(mn)$ time. For consistent networks, d will be a solution to the original STN; otherwise, the algorithm will report that the STN is inconsistent. For the sample STN graph from Figure 1, the single-source version of Bellman-Ford yields the solution: $\{X = -7, A_2 = -2, C_2 = A_1 = -1, C_1 = 0\}$. The single-sink version is similar except that the initial value $d(X) = 0$ represents the presence of an edge from each X to S of length 0 and, for consistent networks, the function $-d$ yields a solution to the original STN. For the sample STN graph, the single-sink version yields the solution: $\{X = A_2 = 0, C_2 = 1, A_1 = 4, C_1 = 7\}$.

Dijkstra's SSSP algorithm. Several of the algorithms presented in this paper use variants of Dijkstra's SSSP algorithm (Cormen et al., 2009), in some

Algorithm 3: $\mathcal{Q}.insertOrDecreaseKeyIfSmaller(X, p)$, a method for a priority queue $\mathcal{Q}$

Input: X, a time-point; p, a priority/key for X
Output: $\top$, unless X has already been popped from the queue with a key greater than p

1 **if** $(\mathcal{Q}.state(X) ==$ `notYetInQ`$)$ **then** $\mathcal{Q}.insert(X, p)$
2 **else if** $((\mathcal{Q}.state(X) ==$ `inQ`$)$ `and` $(p < \mathcal{Q}.key(X)))$ **then** $\mathcal{Q}.decreaseKey(X, p)$
3 **else if** $((\mathcal{Q}.state(X) ==$ `alreadyPopped`$)$ `and` $(p < \mathcal{Q}.key(X)))$ **then** **return** $\perp$
4 **return** $\top$

cases propagating forward from a source node, in other cases propagating backward from a sink node. Pseudocode for a typical implementation of the source-version of Dijkstra's algorithm is given in Algorithm 2. This paper assumes that the implementation of a *priority queue* (e.g., as used by Dijkstra or other algorithms later in the paper) includes the operators shown in Table 1. In addition, it is convenient to define the *insertOrDecreaseKeyIfSmaller* method shown as Algorithm 3. Note that when called from contexts where it is known that the relevant network has no negative loops, the return value for Algorithm 3 may be safely ignored. In other contexts, a return value of $\perp$ indicates that a negative loop was detected (e.g., when trying to update a potential function in response to the insertion of new edges into a graph).

Although Dijkstra's algorithm typically applies only to STNs whose edge-weights are all non-negative, it may also be used for STNs having some negative edges, as long as those negative edges either all emanate from or all terminate in a single time-point (Morris, 2014). In addition, as in Johnson's algorithm (Cormen et al., 2009), a *potential function* can be used to convert the edge-weights in a consistent STN to non-negative values, thereby enabling the use of Dijkstra on the converted graph to guide the traversal of shortest paths in the original graph, as follows.

If h is a solution to an STN $\mathcal{S} = (\mathcal{T}, \mathcal{C})$, then for each constraint $Y - X \leq \delta$ in $\mathcal{C}$, it follows that $h(Y) - h(X) \leq \delta$ which, in turn, implies that $\delta' = h(X) + \delta - h(Y) \geq 0$. Therefore, given an STN graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, the modified STN graph $\mathcal{G}' = (\mathcal{T}, \mathcal{E}')$ obtained by replacing each edge $X \xrightarrow{\delta} Y$ in $\mathcal{E}$ by a corresponding edge $X \xrightarrow{\delta'} Y$ in $\mathcal{E}'$ has only non-negative edges. Figure 2 shows the sample STN from Figure 1 with edge-weights modified to be non-negative using the potential function h corresponding to the solution $\{X = -7, A_2 = -2, C_2 = A_1 = -1, C_1 = 0\}$ seen earlier. Crucially, shortest paths in the modified graph $\mathcal{G}'$ correspond to shortest paths in the original graph $\mathcal{G}$, albeit with different lengths. As a result, Dijkstra's algorithm can be run on $\mathcal{G}'$ to efficiently traverse shortest paths in *both*

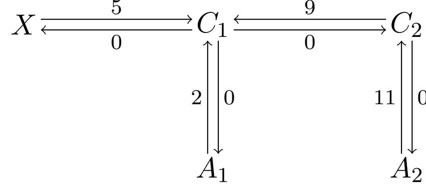


Figure 2: The sample STN graph from Figure 1 with edge-weights modified to be non-negative using the potential function corresponding to the solution $\{X = -7, A_2 = -2, C_2 = A_1 = -1, C_1 = 0\}$

graphs in parallel. In particular, as each edge (X, δ', Y) in $\mathcal{G}'$ is traversed by Dijkstra, the corresponding edge (X, δ, Y) is traversed in $\mathcal{G}$; and each non-negative shortest path-length discovered in $\mathcal{G}'$ is easily convertible into the corresponding shortest path-length in $\mathcal{G}$. For example, the shortest path from X to A_1 has length 5 in $\mathcal{G}'$ in Figure 2; its corresponding length in $\mathcal{G}$ in Figure 1 is computed using the reverse transformation: $-h(X)+5+h(A_1) = -(-7) + 5 + (-1) = 11$. Furthermore, the modified graph $\mathcal{G}'$ need not be constructed at all; instead, for efficiency, the non-negative weights can simply be computed on the fly as they are needed by Dijkstra's algorithm. Finally, potentials can be used in this way to support either a single-*source* or single-*sink* version of Dijkstra.

1.1 Simple Temporal Networks with Uncertainty

A Simple Temporal Network with Uncertainty (STNU) is an STN augmented to include *contingent links* that can be used to represent actions with uncertain durations (Morris, Muscettola, & Vidal, 2001).

Definition 3 (STNU)**.** 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, $A_i, C_i \in \mathcal{T}$ and $0 < x_i < y_i < \infty$; and
  - $C_i \equiv C_j$ iff $i = j$.[1]

  A_i is called the *activation* time-point (ATP) for the ith contingent link; C_i is called its *contingent* time-point (CTP).

It is common practice to let k denote the number of contingent links in an STNU. In addition, for convenience, we let $\mathcal{T}_C = \{C_i \mid \exists (A_i, x_i, y_i, C_i) \in \mathcal{L}\}$ denote the set of contingent time-points; $\mathcal{T}_A = \{A_i \mid \exists (A_i, x_i, y_i, C_i) \in \mathcal{L}\}$ the

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

set of activation time-points; $\mathcal{T}_X = \mathcal{T} \backslash \mathcal{T}_C$ the set of *executable* time-points; and we may refer to each $i \in \{0, 1, \ldots, k-1\}$ as the corresponding *contingent-link index* (CLI). In addition, we let $\Delta_i = y_i - x_i$ equal the duration of the ith contingent link. In practice, a system using an STNU to represent actions and events controls the execution of the executable time-points, but only *observes* the execution of the contingent time-points as they occur in real time.

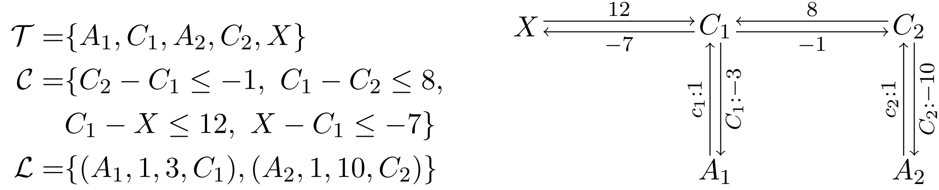


Figure 3: A sample STNU (left) and its corresponding graph (right)

The lefthand side of Figure 3 shows a sample STNU that is similar to the sample STN from Figure 1, except that the intervals from A_1 to C_1 and from A_2 to C_2 are contingent links whose durations are not controlled by the planning system. For this network, the contingent time-points are C_1 and C_2, the activation time-points are A_1 and A_2, and the executable time-points are X, A_1 and A_2.

Definition 4 (STNU graph)**.** Each STNU $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ has a corresponding graph $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, where the time-points in $\mathcal{T}$ serve as the nodes in the graph, and the constraints in $\mathcal{C}$ and the contingent links in $\mathcal{L}$ together correspond to edges in the graph. In particular, $\mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u$ 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 a Simple Temporal Network); and
- for each contingent link (A_i, x_i, y_i, C_i) in $\mathcal{L}$, there is a *lower-case* edge $A_i \xrightarrow{c_i:x_i} C_i$ in $\mathcal{E}_\ell$, and an *upper-case* edge $C_i \xrightarrow{C_i:-y_i} A_i$ in $\mathcal{E}_u$.

The righthand side of Figure 3 shows the STNU graph for the sample STNU from the lefthand side of the figure. The lower-case edges from A_1 to C_1, and from A_2 to C_2 represent the uncontrollable possibilities that the durations of these contingent links may each turn out to be as low as 1. The upper-case edges represent the uncontrollable possibilities that the durations of these contingent links 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); a lower-case edge $A_i \xrightarrow{c_i:x_i} C_i$ as $(A_i, c_i{:}x_i, C_i)$; and an upper-case edge $C_i \xrightarrow{C_i:-y_i} A_i$ as $(C_i, C_i{:}{-y_i}, A_i)$. Since each contingent link has unique lower- and upper-case edges, an algorithm may iterate over the lower- or upper-case edges in

Operator	$\mathcal{G}$	Output/Side Effect
$\mathcal{G}.getOrdEdgeWt(X, Y)$	STN or STNU	Returns the weight of the ordinary edge from X to Y from $\mathcal{G}$ or, if none, ∞.
$\mathcal{G}.insertOrUpdateOrdEdge(X, \delta, Y)$	STN or STNU	If there is no ordinary edge from X to Y in $\mathcal{G}$, inserts the edge (X, δ, Y); if the current edge from X to Y has weight greater than δ, updates the weight to δ.
$\mathcal{G}.getUCEdgeWt(X, A, C)$	STNU	Returns the weight of the UC-edge from X to A labeled by C or, if none, ∞.
$\mathcal{G}.insertOrUpdateUCEdge(X, C{:}w, A)$	STNU	If no UC-edge from X to A labeled by C, inserts edge $(X, C{:}w, A)$; if current UC-edge from X to A labeled by C has weight greater than w, updates weight to w.

Table 2: Operators provided by an implementation of an STN/STNU graph $\mathcal{G}$

an STNU graph simply by iterating over the CLI, $i \in \{0, 1, \ldots, k-1\}$, and fetching the desired edge from the ith contingent link.

Finally, it is convenient to define the following notation (Hunsberger, 2015a):

- $\mathcal{E}_\ell \cup \mathcal{E}_o$, called the *LO-edges,* comprise the lower-case and ordinary edges.
- $\mathcal{G}_{\ell o} = (\mathcal{T}, \mathcal{E}_\ell \cup \mathcal{E}_o)$, called the *LO-graph,* contains only the LO-edges.
- $\mathcal{E}_o \cup \mathcal{E}_u$, called the *OU-edges,* comprise the ordinary and upper-case edges.
- $\mathcal{G}_{ou} = (\mathcal{T}, \mathcal{E}_o \cup \mathcal{E}_u)$, called the *OU-graph,* contains only the OU-edges.

STN/STNU Graph Operations. This paper assumes that the implementation of an STN/STNU graph, $\mathcal{G}$, includes the functionality shown in Table 2.

1.2 Dynamic Controllability of an STNU

As already mentioned, contingent links can be used to represent actions with uncertain durations. A system using an STNU typically controls the execution of the executable time-points, but only *observes* the execution of the contingent time-points as they occur in real time. The most important property of an STNU is called *dynamic controllability* (DC) (Morris et al.,

2001). In short, an STNU $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ is DC if there exists a dynamic strategy for executing the time-points in $\mathcal{T}_X$ that guarantees the satisfaction of all ordinary constraints in $\mathcal{C}$ no matter how the durations of the contingent links in $\mathcal{L}$ turn out in real-time. Crucially, a dynamic strategy can *react* to observations of executions of contingent time-points in real time, but its execution decisions cannot depend on advance knowledge of future contingent executions.[2]

The sample STNU from Figure 3 is not dynamically controllable (i.e., there is no strategy for executing the executable time-points, 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, if the edge from X to C_1 is weakened from 12 to 14, then it is not hard to check that the following strategy for executing X, A_1 and A_2 will work:

> Execute both X and A_2 at time 0.
> If C_2 is observed to execute before time 6, then execute A_1 at time 6.
> Otherwise, execute A_1 at time $C_2 + 1$.

This paper does not repeat the formal definition of the dynamic controllability of an STNU. The reason is that all of the *DC-checking algorithms* from the literature discussed in this paper have been proven to be sound and complete for checking the DC property for STNUs. Their proofs of correctness rely on the Fundamental Theorem of STNUs which states that an STNU is DC if and only if its graph has no *semi-reducible* negative loops (SRN loops) (Morris, 2006; Hunsberger, 2014b). (SRN loops will be defined below.) Each of the algorithms discussed in this paper propagates constraints (equivalently, generates new edges in the STNU graph) to determine whether the graph contains an SRN loop. The algorithms differ not only in the constraint-propagation/edge-generation rules that they employ, but also in their high-level structure (e.g., forward vs. backward propagation, recursive vs. iterative, the need for initializing and updating a potential function and, if needed, the edge-sets on which the potential function is based). As will be seen, these differences can greatly affect the performance of the algorithms. The new algorithm introduced in this paper combines pre-existing and novel techniques to achieve an order-of-magnitude improvement in performance on a variety of benchmark problems compared to the fastest known DC-checking algorithm for STNUs from the literature.

[2]As is common in the literature, this paper and all of the algorithms addressed in this paper assume the version of dynamic controllability in which dynamic strategies may react *instantaneously* to observations of contingent executions (Morris, 2006). As a result, their execution decisions may depend on past *or present* observations.

2 Existing DC-Checking Algorithms for STNUs

Over the past fifteen years, three DC-checking algorithms for STNUs have been presented that have made substantial improvements. The worst-case performance of these algorithms depends on the number of nodes n, the number of contingent links k, and the number of edges m, as follows.

Algorithm (Author(s), Year)	Worst-Case Time-Complexity
Morris, 2006	$O(kmn + k^2n^2 + k^3n)$
Morris, 2014	$O(n^3)$
Cairo, Hunsberger & Rizzi, 2018	$O(mn + k^2n + kn \log n)$

If $k = O(n)$ (e.g., the number of contingent links might be ten or twenty percent of the total number of nodes), then the complexity of Morris' 2006 algorithm simplifies to $O(n^4)$, while that of the other two algorithms simplifies to $O(n^3)$. However, if the number of contingent links is only $k = O(\sqrt{n})$, then the Cairo et al. algorithm out-performs the other two if the number of edges m is significantly less than $O(n^2)$, as follows.

Algorithm (Author(s), Year)	$m = O(n)$	$m = O(n \log n)$	$m = O(n^2)$
Morris, 2006	$O(n^3)$	$O(n^3)$	$O(n^4)$
Morris, 2014	$O(n^3)$	$O(n^3)$	$O(n^3)$
Cairo, Hunsberger & Rizzi, 2018	$O(n^2)$	$O(n^2 \log n)$	$O(n^3)$

The rest of this section describes the operation of these DC-checking algorithms, highlighting the similarities and differences in their approaches. Afterward, a new algorithm is introduced that an empirical evaluation demonstrates out-performs the pre-existing algorithms by at least an order of magnitude.

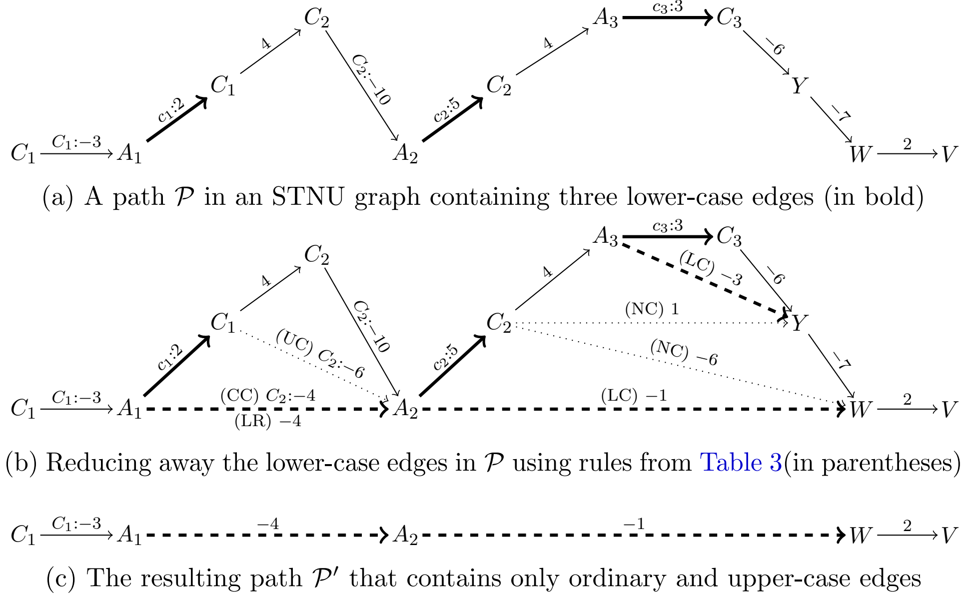
Approaches to DC-checking algorithms in the literature. Each of these algorithms is based on a set of rules that can be equivalently viewed as constraint-propagation rules, edge-generation rules, or path-transformation rules. Each algorithm focuses on using its particular set of path-transformation rules to effectively *reduce away* certain kinds of edges—let's call them *problematic edges*—(i.e., to transform paths into new paths that do not contain the problematic edges). Each algorithm repeatedly uses some variant of Dijkstra's algorithm to traverse some subset of paths in the STNU graph, looking for opportunities to reduce away the problematic edges. The algorithms differ in the kinds of edges they view as problematic (i.e., that they seek to reduce away), the sets of edge-generation rules they employ, the particular variants of Dijkstra's algorithm that they use (some of which depend on computing or updating potential functions to re-weight edges in the graph), and the direction in which paths are traversed.

Rule	Graphical Representation	Applicability Conditions
No Case (NC)	$X \xrightarrow{v} Y \xrightarrow{w} W$, generated $X \overset{v+w}{\dashrightarrow} W$	(none)
Upper Case (UC)	$X \xrightarrow{v} Y \xrightarrow{C:w} A$, generated $X \overset{C:v+w}{\dashrightarrow} A$	(none)
Lower Case (LC)	$A \xrightarrow{c:x} C \xrightarrow{w} X$, generated $A \overset{x+w}{\dashrightarrow} X$	$w < 0$
Cross Case (CC)	$A \xrightarrow{c:x} C \xrightarrow{K:w} B$, generated $A \overset{K:x+w}{\dashrightarrow} B$	$K \not\equiv C$, $w < 0$
Label Removal (LR)	$X \xrightarrow{C:w} A \xrightarrow{c:x} C$, generated $X \overset{w}{\dashrightarrow} A$	$w \geq -x$

Table 3: Edge-generation rules from Morris and Muscettola (2005)

2.1 Morris' 2006 $O(n^4)$ DC-Checking Algorithm

Morris' 2006 algorithm is based on the edge-generation/constraint-propagation/path-trans-formation rules listed in Table 3 (Morris, 2006). The approach taken by the algorithm is to transform paths in the STNU graph into paths that have only ordinary or upper-case edges by *reducing away* its lower-case edges. The approach is illustrated in Fig. 4.



(a) A path $\mathcal{P}$ in an STNU graph containing three lower-case edges (in bold)

(b) Reducing away the lower-case edges in $\mathcal{P}$ using rules from Table 3(in parentheses)

(c) The resulting path $\mathcal{P}'$ that contains only ordinary and upper-case edges

Figure 4: Transforming a path $\mathcal{P}$ in the STNU graph into a path $\mathcal{P}'$ in the OU-graph by *reducing away* the lower-case edges in $\mathcal{P}$

The path $\mathcal{P}$ in Fig. 4a contains ordinary, lower-case, and upper-case edges. Fig. 4b shows how the rules from Table 3 can be applied to the edges in $\mathcal{P}$

to effectively transform it into a path $\mathcal{P}'$, shown in Fig. 4c, that contains only OU-edges. Paths like $\mathcal{P}$, that can be transformed using the rules from Table 3 into paths that contain only OU-edges, are called *semi-reducible*. (Note that the rules in Table 3 only generate ordinary or upper-case edges, never lower-case edges.)

Morris' analysis of semi-reducible paths yielded several important results, the most important being that an STNU is dynamically controllable if and only if it contains no semi-reducible negative (SRN) loops. As a result, a DC-checking algorithm need only focus on determining whether an STNU graph contains any SRN loops. Second, he proved that the paths used to reduce away lower-case edges—what he called *extension sub-paths*—have several important properties that can be leveraged by a DC-checking algorithm to make it more efficient.

For illustration, return to Fig. 4. Each lower-case edge in the semi-reducible path $\mathcal{P}$ in Fig. 4a has a corresponding extension sub-path that is a sub-path of the original path $\mathcal{P}$. For example, the extension sub-path used to reduce away the lower-case edge $(A_1, c_1{:}2, C_1)$ is the two-edge path from C_1 to C_2 to A_2. Sequential applications of the UC, CC and LR rules generate the edge $(A_1, -4, A_2)$, drawn as a bold, dashed arrow in Fig. 4b.[3]

Similarly, the extension sub-path used to reduce away the lower-case edge $(A_3, c_3{:}3, C_3)$ is the single edge $(C_3, -6, Y)$. An application of the LC rule generates the edge $(A_3, -3, Y)$, also drawn as a bold, dashed edge. Finally, and most interestingly, the extension sub-path used to reduce away the lower-case edge $(A_2, c_2{:}5, C_2)$ is the four-edge path from C_2 to A_3 to C_3 to Y to W. Note that this extension sub-path *fully contains* the extension sub-path for $(A_3, c_3{:}3, C_3)$ (i.e., the extension sub-path for $(A_3, c_3{:}3, C_3)$ is *nested inside* the extension sub-path for $(A_2, c_2{:}5, C_2)$). Because of this nesting, the lower-case edge $(A_3, c_3{:}3, C_3)$ must be reduced away first, before subsequent edge-generation rules may be applied to reduce away $(A_2, c_2{:}5, C_2)$, eventually generating the edge $(A_2, -1, W)$, again drawn as bold and dashed.

Morris showed that although extension sub-paths have negative length, every non-empty proper prefix of an extension sub-path has non-negative length. Furthermore, he proved that for the purposes of DC checking, it suffices to restrict attention to *breach-free* extension sub-paths that are nested to a depth of at most k.[4] As a result, when searching for extension sub-paths that can be used to reduce away a lower-case edge $(A_i, c_i{:}x_i, C_i)$, the algorithm restricts attention to shortest *allowable* paths emanating from C_i,

[3]The intermediate edges, drawn as dotted arrows in the figure, are important only as stepping stones to generating the bold, dashed edge that reduces away the lower-case edge. As will be seen, Morris' DC-checking algorithm need only insert the bold, dashed edges into the STNU graph to achieve the desired result.

[4]An extension sub-path for a lower-case edge $(A, c{:}x, C)$ is breach-free if it does *not* contain an occurrence of any upper-case edge $(W, C{:}{-w}, A)$ associated with the same contingent link.

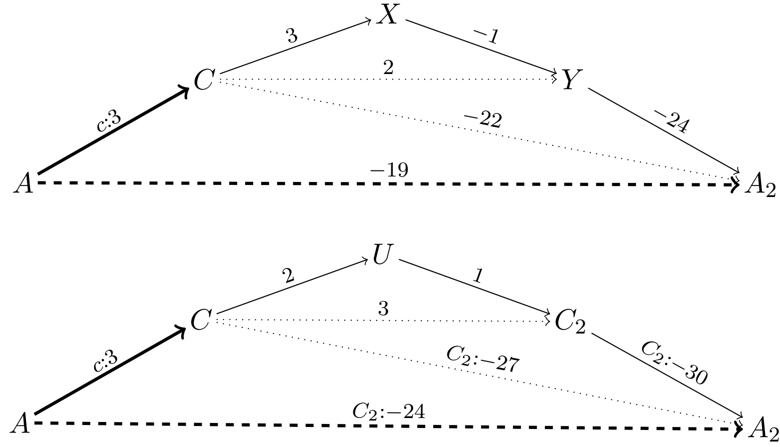


Figure 5: Generating different edges from A to A_2, both of which must be kept

where an *allowable* path (for C_i) is any breach-free path in the OU-graph whose non-empty proper prefixes have non-negative length.

However, one additional issue arises. For example, consider the exploration of extension sub-paths emanating from a contingent time-point C, looking for ways to reduce away the lower-case edge $(A, c{:}3, C)$. It can happen, as shown in Figure 5, that two different extension sub-paths from C to a time-point A_2 lead to the generation of different edges, one ordinary and one upper-case, with different lengths, both of which must be kept by the algorithm. As this example illustrates, it is insufficient to restrict attention to shortest breach-free paths emanating from C in the OU-graph. However, although Morris didn't explicitly elaborate a solution to this issue, it can be handled as follows. First, for the generation of ordinary edges, use a potential function to enable a Dijkstra-like traversal along shortest breach-free paths emanating from C in the OU-graph, but restricted to paths that are either (1) non-negative, or (2) negative but would lead to the generation of an ordinary edge. Second, while following such paths, if propagating forward along an upper-case edge would immediately terminate an extension sub-path leading to the generation of a UC edge, then generate that edge.

Pseudo-code for Morris' 2006 algorithm is given as Algorithm 4. The algorithm has the following features:

- An *outer loop* (Lines 1–31) of k iterations. The ith iteration of the outer loop generates all edges that can be obtained by reducing away lower-case edges whose extension sub-paths are nested to a depth of at most i.

- Each iteration of the outer loop contains an *inner loop* (Lines 5–28), also of k iterations. The jth iteration of the inner loop processes the jth

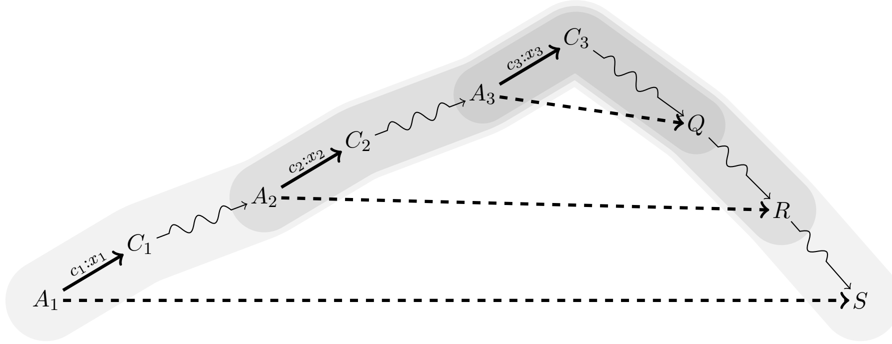


Figure 6: Reducing away three nested lower-case edges in an STNU graph

lower-case edge (A_j, x_j, y_j, C_j) by exploring shortest allowable paths emanating from the contingent time-point C_j in the OU-graph as described earlier.

- The exploration of extension sub-paths emanating from a contingent time-point C in the OU-graph is guided by Dijkstra's algorithm using a priority queue $\mathcal{Q}$ initialized to include just C (Lines 6-7). Since Dijkstra's algorithm only works on graphs with non-negative edges, each iteration of the outer loop starts (at Line 2) by running the Bellman-Ford algorithm (Algorithm 1) on the OU-graph to generate a solution that can be used as a potential function, h, to effectively transform the edge weights in the OU-graph into non-negative values. (By ignoring the alphabetic labels on the upper-case edges, the OU-graph can be viewed as an STN.) As in Johnson's algorithm, Dijkstra can then be used to guide an efficient traversal of shortest paths in the OU-graph. For efficiency, the re-weighting of edges in the OU-graph is done on the fly, as edges are processed (e.g., at Lines 10, 16 and 22), without creating a new graph structure. On the other hand, if Bellman-Ford determines that there is no solution, then the STNU must be non-DC (Line 3).

Fig. 6 is a sketch of a semi-reducible path that contains multiple occurrences of lower-case edges, and whose extension sub-paths are nested.[5] For example, the extension sub-path from C_3 to Q that is used to reduce away the lower-case edge $A_3 \xrightarrow{c_3:x_3} C_3$ is nested within the extension sub-path from C_2 to R that is used to reduce away the lower-case edge $A_2 \xrightarrow{c_2:x_2} C_2$ which, in turn, is nested within the extension sub-path from C_1 to S that is used to reduce away the lower-case edge $A_1 \xrightarrow{c_1:x_1} C_1$.

If, during the inner loop of Morris' algorithm, the lower-case edges happen to be processed in the order $(A_1, c_1{:}x_1, C_1)$, $(A_2, c_2{:}x_2, C_2)$, $(A_3, c_3{:}x_3, C_3)$,

[5] To decrease clutter, the intermediate edges, drawn as dotted arrows in Fig. 4b, are not shown in Fig. 6.

Figure 7: Morris' 2006 algorithm discovering a semi-reducible negative loop in the STNU graph from Figure 3

then the first iteration of the outer loop will only generate the bold, dashed edge from A_3 to Q; the second iteration will generate the bold, dashed edge from A_2 to R; and the third iteration will generate the bold, dashed edge from A_1 to S. Since the algorithm does not know in advance in which order to process the lower-case edges, it must do k iterations of the outer loop to be sure that any nestings up to depth k can be effectively reduced away. Furthermore, since inserting new edges into the graph typically requires updating the potential function, new edges are *accumulated* during the k iterations of the inner loop, but are only *inserted* into the graph at the end of each iteration of the outer loop (Lines 30-31). The potential function is then updated by running Bellman-Ford at the beginning of the next iteration of the outer loop (Line 2).

After k iterations of the outer loop, the algorithm makes a final call to Bellman-Ford on the modified OU-graph to determine whether the network is dynamically controllable (Line 32). Overall, the algorithm runs Bellman-Ford at most $k + 1$ times[6] and does at most k^2 Dijkstra-like traversals, each involving at most $m + kn$ edges, resulting in an overall complexity of:

$$O(k(m + kn)n + k^2((m + kn) + n \log n)) = O(kmn + k^2n^2 + k^3n)$$

For dense graphs, where $m = O(n^2)$ and $k = O(n)$, this complexity reduces to $O(n^4)$. However, for very sparse graphs where $m = O(n)$ and $k = O(\sqrt{n})$, it reduces to $O(n^3)$.

Fig. 7 illustrates how Morris' 2006 algorithm discovers an SRN loop in the sample STNU graph seen previously in Figure 3. The depicted loop travels around the perimeter of the figure, visiting C_1 four times, and A_1 and C_2 twice each. It is called an *indivisible* SRN loop because none of its many subsidiary loops are themselves SRN loops (Hunsberger, 2013, 2014b). The

[6]If an early iteration of the outer loop does not generate any new edges, then the algorithm can terminate early (Line 29).

edges generated by Morris' algorithm are dashed in the figure. They are used to transform the original loop into the loop $(A_1, 0, C_2, C_2{:}{-10}, A_2, 0, X, 9, A_1)$ which contains only OU-edges, and has length -1, signaling that the STNU is not DC.

Note that if the edge in the original STNU graph from C_1 to X were weakened from 12 to 14, then the length of the loop would be 1 (i.e., not negative). Indeed, in that case, the STNU would be dynamically controllable, as evidenced by the dynamic execution strategy that was given for it earlier.

Hunsberger's speed-ups of Morris' 2006 algorithm. Hunsberger (2013, 2014b) showed that for each non-DC network, there is at least one *nesting order* (equivalently, one order in which to process the lower-case edges) that will make Morris' algorithm generate a negative loop in the OU-graph—and hence a non-DC answer—in precisely *one* iteration of the outer loop. If one of those nesting orders could be determined in advance, then the modified algorithm would run in $O(n^3)$ time. Hunsberger showed that using a random order works well in practice; he also presented an $O(n^3)$ heuristic for selecting a "good" order. An empirical evaluation showed that this heuristic resulted in improved performance. For the network in Figure 7, the modified algorithm would perform two iterations of the outer loop if the lower-case edges are explored in the order $(A_2, 1, 10, C_2), (A_1, 1, 3, C_1)$, but would require only one iteration if the opposite order were followed.

Subsequently, Hunsberger (2014a, 2015b) made some additional modifications to Morris' algorithm to further improve its performance. First, instead of waiting until the end of each iteration of the *outer* loop to insert new edges, it inserts new edges after each iteration of the *inner* loop. Second, since inserting new edges requires updating the potential function, any time an iteration of the inner loop inserts new edges, it immediately updates the potential function. Third, because the processing of a lower-case edge $(A, c{:}x, C)$ can only generate edges emanating from A, it is able to efficiently update the potential function using a technique called *rotating Dijkstra,* as follows.[7]

First, at the beginning of each iteration of the inner loop, when preparing to process a lower-case edge—say $(A, c{:}x, C)$—the following invariant holds: for each $X \in \mathcal{T}$, the potential function h satisfies $h(X) = -\mathcal{D}(X, A)$, where $\mathcal{D}(X, A)$ is the length of the shortest path from X to A.[8] Since inserting edges emanating from A cannot change the lengths of any shortest paths terminating at A, h remains a solution—unless a negative loop has been introduced. Therefore, in preparation for processing the next lower-case edge—say, $(A', c'{:}x', C')$—the potential function h can be used to guide a

[7] Ramalingam et al. (1999) used similar techniques to update potential functions.

[8] For the first iteration, the initial potential function h is extracted from a preliminary run of Johnson's algorithm. If that were the only goal, a *single-sink* version of Bellman-Ford would have been more efficient.

separate, single-sink version of Dijkstra, with A' as the sink, to compute a new potential function h' that, in the absence of a negative loop, satisfies $h'(X) = -\mathcal{D}(X, A')$ for each $X \in \mathcal{T}$, thereby restoring the invariant. By inserting new edges after each inner iteration, subsequent inner iterations can propagate along the newly found edges, making it possible for the algorithm to frequently terminate earlier than it otherwise would. In particular, if any k consecutive iterations of the inner round—even if not all contained within one outer iteration—fail to generate any new edges, then the algorithm can immediately halt, declaring the network to be DC.

Although additional modifications to Morris' 2006 algorithm might be able to incrementally improve its performance even further, completely different approaches to the DC-checking problem for STNUs, presented below, turn out to offer much greater speed-ups.

2.2 Morris' 2014 DC-Checking Algorithm

Morris (2014) subsequently introduced a new DC-checking algorithm, based on the same edge-generation rules as his 2006 algorithm, but with the following significant differences:[9]

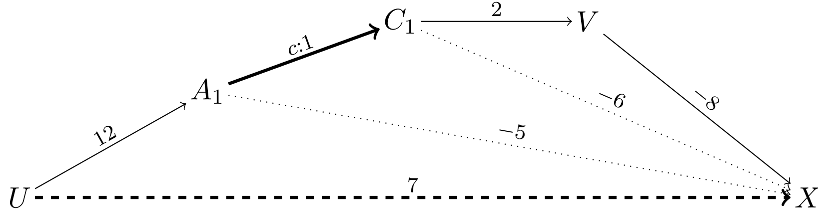
- It aims to reduce away *negative* edges, whether ordinary or upper-case.
- It propagates *backward,* along incoming *non-negative* edges, whether ordinary or lower-case.
- It does not need a potential function and hence does not need to call Bellman-Ford.
- It only inserts non-negative, ordinary edges into the STNU graph.

Note that a negative edge in an STNU graph is necessarily either an ordinary edge or an upper-case edge. For Morris' 2014 algorithm, each time-point that has at least one incoming negative edge is called a *negative node.* The algorithm processes each negative node, back-propagating along paths in the LO-graph (i.e., the graph consisting of ordinary or lower-case edges), looking for opportunities to reduce away the negative edges.[10]

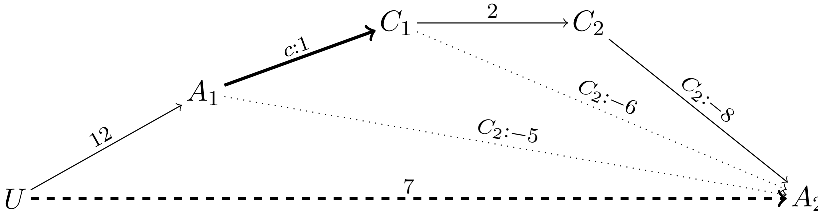
One advantage of back-propagating along non-negative edges (even though the initial edge is negative) is that Dijkstra's algorithm can be used without requiring a potential function to reweight any edges. For example, suppose the time-point X has three incoming negative edges: $(U, -5, X), (V, -8, X)$ and $(W, -12, X)$. The algorithm initializes its priority queue with U, V and

[9] The same year, Nilsson et al. (2014) independently presented a cubic algorithm for the *incremental* DC-checking problem that has been conjectured to work also for the full DC-checking problem.

[10] An edge E is *suitable* for back-propagation from X *unless* X is a contingent time-point and E is the lower-case edge for X.



(a) Reducing away the negative edge $(V, -8, X)$, generating the non-negative edge $(U, 7, X)$



(b) Reducing away the upper-case edge $(C_2, C_2{:}{-8}, A_2)$, generating the edge $(U, 7, A_2)$

Figure 8: Reducing away negative edges by the Morris 2014 DC-checking algorithm

W having the priorities -5, -8 and -12, respectively. It then does a single-*sink* shortest-paths variant of Dijkstra, back-propagating along non-negative edges in the LO-graph.

Figure 8 gives examples of how Morris' 2014 algorithm reduces away negative edges, whether ordinary or upper-case. Note that the dotted, *negative* edges—some of which may be upper-case edges—need not be inserted into the graph; these intermediate edges are only stepping stones to the dashed, *non-negative* edge which, due to the Label-Removal rule, is guaranteed to be an ordinary edge. The non-negative ordinary edge is the one that reduces away the original negative edge.

If, during its processing of some negative node X, the back-propagation along non-negative edges ever encounters another negative edge—say, an edge $(D, -15, E)$—then it suspends its processing of X and instead processes the newly encountered negative node E. There are two possible outcomes: (1) the processing of E completes (i.e., all negative edges incoming to E can be reduced away), in which case the processing of X can continue; or (2) the processing of E is interrupted by the processing of some other negative node F, whose processing in turn is interrupted by some other negative node G, and so on, until a cycle of such interruptions is encountered, which signals a negative loop in the OU-graph.

Since the algorithm only back-propagates along non-negative edges, no potential function is needed; hence the algorithm does not need to call

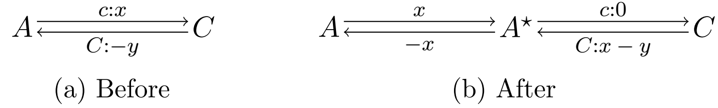


Figure 9: Converting a contingent link into *normal form* for Morris' 2014 algorithm

Bellman-Ford at all. Furthermore, because a cycle of interruptions necessarily implies the existence of a negative loop, the algorithm does not need to explicitly worry about the nesting order that caused Morris' 2006 algorithm to do k iterations of its outer loop.[11] Since there are at most n time-points with incoming negative edges, and the processing of each such time-point involves a Dijkstra-like traversal that can add up to n new edges, there can be up to n^2 new edges inserted overall, leading to an overall complexity of $O(n^3)$.

Converting an STNU into Normal Form. Morris' 2014 algorithm relies on the following property: each negative node has either (1) exactly one incoming negative edge, which happens to be an upper-case edge; or (2) one or more incoming negative edges, all of which are ordinary. To see why, suppose that a negative node X happened to be an activation time-point for two different contingent links. Then X would have two incoming UC edges, say, one labeled by C and the other labeled by C', which would require the algorithm to distinguish intermediate edges labeled by C from those labeled by C'. Similar remarks apply to the case where a negative node has incoming negative ordinary edges and an incoming upper-case edge. To avoid the complexities raised by such cases, Morris' 2014 algorithm first does an $O(k)$-time pre-processing step that ensures that the above-mentioned property holds for each negative node. The pre-processing step converts each contingent link (A, x, y, C) into what Morris called its *normal form,* $(A^\star, 0, y{-}x, C)$, where $A^\star$ is a new time-point that is inserted into the network and constrained to occur exactly x units after A, as illustrated in Figure 9. After converting all contingent links in this way, each new time-point $A^\star$ is a negative node having exactly one incoming negative edge which happens to be a UC edge, and all other negative nodes have only ordinary incoming negative edges.[12] The computation time required to convert the contingent links into their normal form is completely negligible compared to the overall execution time of the algorithm.

[11] In contrast, if Morris' 2006 algorithm interrupted its forward processing of a lower-case edge each time it encountered another lower-case edge, any cycle of such interruptions would only yield a *positive* loop, which does not help to answer the question of DC versus non-DC.

[12] Thanks to Andrei Stanciu for pointing out that this conversion, which is simpler than the one given by Morris, yields the desired property.

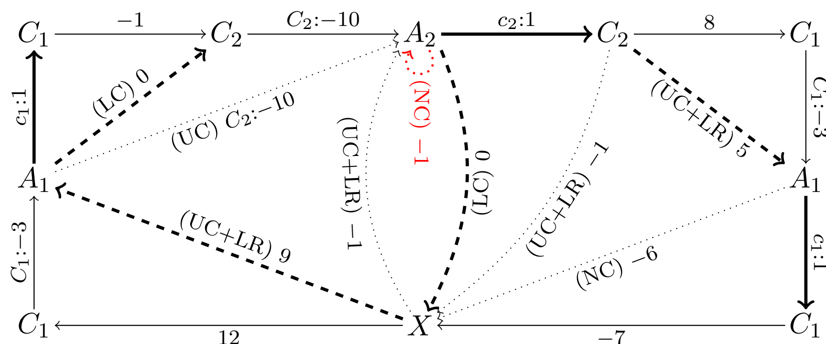


Figure 10: Morris' 2014 algorithm discovering a semi-reducible negative loop by generating a negative loop in the OU-graph for the sample STNU from Figure 3

The high-level loop of Morris' 2014 DC-checking algorithm, which iterates through the negative nodes, is given as Algorithm 5; the recursive helper function, `DCbackprop`, which performs the back-propagation from a given negative node, is given as Algorithm 6. When called on a negative node X, the `DCbackprop` function first checks (Line 1) whether there is a prior call to `DCbackprop` for the same node X on the recursive function-call stack that has not yet finished processing, which would signal a negative loop.[13] Next (Line 2), it checks whether a prior call had completed its processing of X, which would obviate the need for further processing. The rest of the code (Lines 3–23) handles the actual back-propagation from X. Lines 3–12 initialize a priority queue, as follows. For each negative edge (Y, δ, X) terminating at X, it inserts the node Y with the key δ. (If the edge is an upper-case edge, the upper-case label is ignored.) Lines 13–23 run Dijkstra's algorithm on edges in the LO-graph, with some slight modifications. If the distance, $\mathtt{dist}[U]$ from the current node U to X is non-negative (Line 15), it inserts the edge $(U, \mathtt{dist}[U], X)$ into the graph and does no further back-propagation from U. However, if $\mathtt{dist}[U]$ is negative, it continues with Dijkstra-like back-propagation from U (Lines 17–22). First (Line 17), it checks whether U is a negative node and, if so, makes a recursive call to `DCbackprop` on U to ensure that all negative edges terminating at U have been reduced away. Next (Lines 18–22), it back-propagates along all non-negative LO-edges that terminate at U, with only one exception: if X happens to be an activation time-point for an upper-case edge $(C, C{:}-y, X)$, and $U \equiv C$, it does not back-propagate along the lower-case edge $(X, c{:}x, C)$ (Line 19) due to the restriction in the Cross Case rule (cf. Table 3).[14]

[13] As Morris noted, a vector that keeps track of the status of each negative node (e.g., `unstarted`, `started` or `finished`) can make the checks in Lines 1 and 2 of `DCbackprop` efficient.

[14] In the pseudocode, the condition $V \equiv X$ is equivalent to our description of $U \equiv C$.

Fig. 10 illustrates how Morris' 2014 algorithm processes the loop from Fig. 7, leading to the generation of a negative self-loop, which implies that the STNU is not dynamically controllable. It is easiest to describe the operation of the algorithm on this loop assuming that the negative nodes are processed in the order: C_2, A_1, X, A_2. Back-propagation from the negative node C_2, immediately generates the non-negative edge $(A_1, 0, C_2)$ by an application of the Lower-Case rule to the two-edge path $(A_1, c_1{:}1, C_1, -1, C_2)$. Next, back-propagating from A_1—which appears in twice in the loop, but of course only once in the STNU—generates the edges, $(C_2, 5, A_1)$ and $(X, 9, A_1)$, as follows: the Upper-Case and Label-Removal rules applied to the two-edge path $(C_2, 8, C_1, C_1{:}{-3}, A_1)$ generate the edge $(C_2, 5, A_1)$; and the same rules applied to the two-edge path $(X, 12, C_1, C_1{:}{-3}, A_1)$ generate the edge $(X, 9, A_1)$.

Next, back-propagating from X along the path $(A_2, c_2{:}1, C_2, 5, A_1, c_1{:}1, C_1, -7, X)$, which includes some of the recently generated (dashed) edges, reduces that path, after several rule applications, to the non-negative edge $(A_2, 0, X)$. Finally, back-propagating from A_2 along the *loop* $(A_2, 0, X, 9, A_1, 0, C_2, C_2{:}{-10}, A_2)$, which also includes some of the recently generated (dashed) edges, reduces that loop to the *self-loop* $(A_2, -1, A_2)$, which indicates that the network is not DC.

Incidentally, if the algorithm happened to initially process the negative nodes in some different order, it would yield the same result, because, for example, if back-propagating from X bumped into the negative node A_1, then the processing of X would be suspended until the processing of A_1 was carried out.

2.3 The RUL$^-$ DC-Checking Algorithm

Cairo, Hunsberger and Rizzi (2018) introduced a new DC-checking algorithm for STNUs called the RUL$^-$ algorithm. The letters R, U and L are abbreviations for the three rules used by the algorithm; the minus sign was used to distinguish this algorithm, which propagates lower bounds, from an earlier version that propagates upper bounds. The rules are shown in Table 4, albeit in the order R, L, U for reasons that will become clear later on. A quick glance at the rules used by the RUL$^-$ algorithm reveals that they are related to the rules used by Morris' 2006 and 2014 algorithms; however, there are several important differences:

- The RUL$^-$ rules only generate ordinary edges.
- The Relax$^-$ rule is the same as the No-Case rule from Table 3, except that it has stricter applicability conditions. (Note that $\Delta_i = y_i - x_i$ is the duration of the ith contingent link (A_i, x_i, y_i, C_i).)
- The Lower$^-$ rule is similar to the Lower-Case rule from Table 3,

except that it has *different* applicability conditions: some stricter, some less so.

- The Upper$^-$ rule is similar to the Upper-Case rule from Table 3, except that it is *not* length preserving when $v - y < -x$.

- Since none of the rules ever generate new upper-case edges, the Label-Removal and Cross-Case rules from Table 3 are inapplicable.

The RUL$^-$ algorithm has the following features:

- It views the k *upper-case* edges as the problematic edges that need to be reduced away.

- It propagates backward from each of the k original upper-case edges along lower-case and ordinary edges, including negative ordinary edges.

- It computes and iteratively updates a potential function based on the lower-case and ordinary edges; it then uses that potential function to guide the Dijkstra-like traversal during the backward propagation.

- After processing each upper-case edge, the potential function is updated using a secondary Dijkstra-like traversal.

- Unlike Morris' 2014 algorithm, the RUL$^-$ algorithm does not use recursive function calls to deal with interrupted processing of upper-case edges because the processing of the interrupting upper-case edge typically generates new edges, which requires updating the potential function which, in turn, requires restarting the interrupted processing of the original upper-case edge from scratch, not from where it was interrupted. To deal with this, the algorithm uses two stacks to keep track of the processing status of each of the upper-case edges.

- The improved worst-case complexity comes from the stricter applicability conditions on the *RUL* rules, and the fact that there are only k upper-case edges, where k is usually much smaller than n.

Fig. 11 illustrates how the RUL$^-$ algorithm uses the rules from Table 4 to reduce away the upper-case edge $C \xrightarrow{C:-20} A$ associated with the contingent link $(A, 8, 20, C)$. First, the algorithm back-propagates from C using the Relax$^-$ and Lower$^-$ rules generating the dashed edges at the top of the figure. This propagation continues until the length of the generated edge is greater than or equal to $\Delta = y - x = 20 - 8 = 12$, because the Relax$^-$ and Lower$^-$ rules are only applicable when the length of the righthand edge is less than Δ. In the figure, this happens when the edge $T \xrightarrow{14} C$ is generated, since $14 \geq 12$. Next, for every newly generated (dashed) edge E, the algorithm applies the Upper$^-$ rule to E and the upper-case edge

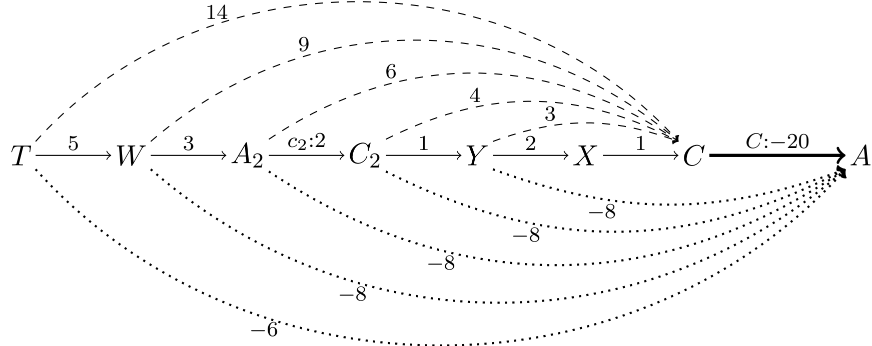
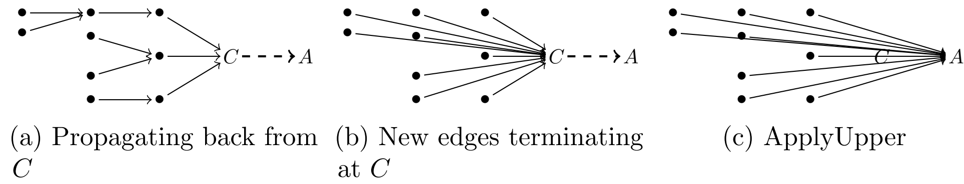


Figure 11: Reducing away the upper-case edge for the contingent link $(A, 8, 20, C)$ using the RUL$^-$ algorithm



(a) Propagating back from C (b) New edges terminating at C (c) ApplyUpper

Figure 12: Reducing away an upper-case edge from C to A with the RUL$^-$ algorithm

$C \xrightarrow{C:-20} A$, generating the dotted edges at the bottom of the figure, each of which terminates at A. Together, these dotted edges represent all of the ways that the RUL$^-$ algorithm reduces away the upper-case edge $C \xrightarrow{C:-20} A$.

An important property of the generated edges terminating at C is that their lengths are less than $\Delta = y - x = 20 - 8 = 12$, except for the edge from T to C, whose length is 14. (That was why the backward propagation stopped at T.) As a result, the lengths of all the generated edges terminating at A are equal to $-8 = -x$, except for the edge from T to A, whose length is $14 - 20 = -6 \geq -8 = -x$. In particular, the non-length-preserving case of the Upper$^-$ rule applies to all of the generated edges terminating at A, except for the edge from T to A. As illustrated in Fig. 12, there may be many different paths terminating at the contingent time-point C. Nonetheless, the process described above still applies. In the first phase, the algorithm uses the Relax$^-$ and Lower$^-$ rules to back-propagate from the contingent time-point C, generating new edges terminating at C; and in the second phase, it applies the Upper$^-$ rule to each of the edges terminating at C to generate edges terminating at A, thereby reducing away the upper-case edge.

Pseudocode for the RUL$^-$ DC-checking algorithm, called RUL-DC-check,

is given in Algorithm 7. Helper functions for computing and updating a potential function are given in Algorithm 8. Helper functions for carrying out the backward propagations described in Figures 11 and 12 are given in Algorithm 9.

Line 1 of the RUL-DC-check algorithm applies a *single-sink* version of the Bellman-Ford algorithm to the $\mathcal{G}_{\ell o}$ graph (i.e., the STN graph consisting of the lower-case and ordinary edges, ignoring alphabetic labels on the LC-edges) to compute a lower-bound potential function h. This version of Bellman-Ford, given in Algorithm 8, is equivalent to first computing for each node X the distance from X to a simulated sink node S, where the distance function d is initialized by $d(X) = 0$ for each X, and then defining the *lower-bound* potential function h by setting $h(X) = -d(X)$ for each X. The given pseudocode follows Cairo et al. (2018) by doing all computations in terms of h, rather than $d = -h$. If Bellman-Ford fails to generate a potential function for the LO-graph, then the algorithm immediately halts (Line 2), signaling that the input STNU is not DC.

Next, Lines 3–5 initialize two stacks: $\mathcal{U}$ and S. The $\mathcal{U}$ stack—$\mathcal{U}$ for *unfinished*—contains the contingent time-points corresponding to upper-case edges that the algorithm has not yet finished processing, whether it has started processing them or not; therefore, initially, $\mathcal{U}$ contains all of the contingent time-points. The S stack—S for *started*—contains the contingent time-points whose corresponding upper-case edges the algorithm has started processing. In effect, S is like a function-call stack for the algorithm. Initially, S contains a single, arbitrarily chosen contingent time-point, C_i. If, during its processing of the upper-case edge for C_i, the algorithm encounters the upper-case edge for some C_j, then it would interrupt its processing of C_i to initiate its processing of C_j, represented by pushing C_j onto the stack. In general, the top of the S stack holds the contingent time-point C for the upper-case edge that the algorithm is currently focused on; the sequence of contingent time-points below C in the stack represent the sequence of processing attempts that were interrupted and are awaiting re-processing.

The `while` loop in Lines 6–20 does the main work of the RUL-DC-check algorithm. Each iteration begins by attempting to process the upper-case edge for the contingent time-point C at the top of the S stack (Line 7). (It does not pop C off the stack yet.) The CloseRelaxLower and ApplyRelaxLower helper functions, defined in Algorithm 9, carry out the backward propagations from C along lower-case and ordinary edges (Lines 8-9) to generate (and insert) new edges terminating at C, as illustrated previously in Figures 11 and 12ab; and the ApplyUpper helper function, also defined in Algorithm 9, applies the UPPER^- rule to each edge terminating at C to generate (and insert) new edges terminating at A (Line 9), thereby effectively reducing away the upper-case edge $(C, C{:}-y, A)$, as illustrated in Figures 11 and 12c.

Since the RELAX^- and LOWER^- rules used by CloseRelaxLower and ApplyRelaxLower are length-preserving rules involving only ordinary and

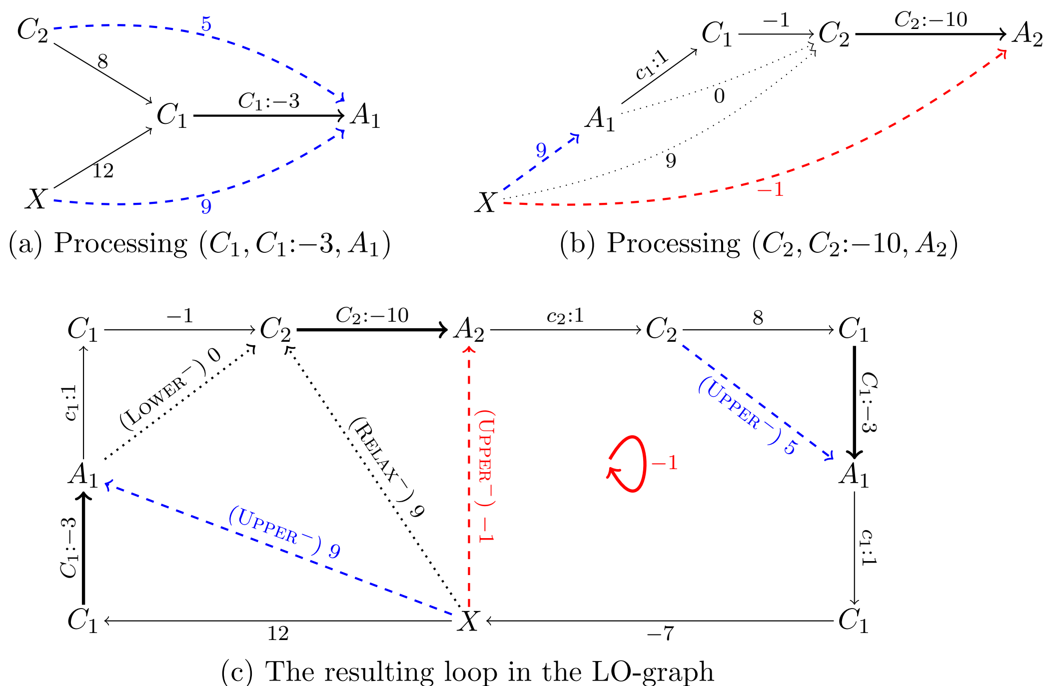
lower-case edges, the edges they generate cannot introduce *shorter* paths in the LO-graph $\mathcal{G}_{\ell o}$ and, hence, cannot disturb the validity of the potential function h. However, the UPPER$^-$ rule used by ApplyUpper involves the upper-case edge $(C, C{:}{-y}, A)$ and, hence, can generate new *shorter* paths in the LO-graph and, thus, can disturb the validity of the potential function. Therefore, the UpdatePotential function, defined in Algorithm 8, updates the potential function h to reflect these newly inserted edges (Line 10).

Just as Hunsberger's speed-up of Morris' 2006 algorithm exploited the fact that all new edges inserted by one iteration of the inner loop of that algorithm necessarily *emanate* from a single time-point to enable a separate *single-sink* version of Dijkstra to efficiently update the potential function for the OU-graph, the UpdatePotential function in Algorithm 8 exploits the fact that all new (shorter) edges inserted by one iteration of the RUL$^-$ algorithm necessarily *terminate* at a single time-point to efficiently update the potential function for the LO-graph. Although it also uses a single-sink version of Dijkstra, the UpdatePotential function, which is equivalent to an incremental update function introduced by Ramalingam et al. (1999), does not do a full run of Dijkstra, but rather only enough to modify h to restore its being a solution. In particular, each node X in the priority queue has as its key the amount by which the potential function $h(X)$ must change to restore h being a solution. If the key of a node X ever reaches zero, then propagation can stop at that point because the potential function does not need to change. Our implementation of the UpdatePotential function improves on the original version in Cairo et al. (2018) by including a more efficient check for a negative loop (Lines 19–20).[15]

The `if` statement (Lines 12–14) of the RUL-DC-check algorithm (Algorithm 7) checks whether any *other* upper-case edge $(C', C'{:}{-y'}, A')$ was encountered during the backward propagation done by CloseRelaxLower that would require interrupting the processing of C. An interruption would be necessary if there were some $C' \in \mathcal{U}$ (i.e., the algorithm has not yet completed its processing of C') and an edge (whether pre-existing or just added) from A' to C of some length less than $\Delta_C = y - x$. Now, if the algorithm had already started its processing of C' during some earlier iteration (i.e., if $C' \in S$), that would signal a cycle of interruptions, and hence a negative loop (Line 13). Otherwise, C' is pushed onto S, with C below C', signaling that C' is interrupting C (Line 14).

The `else` statement (Lines 15–19) handles the case where the processing of C need not be interrupted, in which case, the algorithm has completed its processing of C, signaled by C being removed from both $\mathcal{U}$ and S (Lines 16–17). Now, if S is non-empty, the next iteration will deal with the contingent

[15]This kind of check for a negative cycle was also done by Ramalingam et al. (1999); however their algorithm only considered the insertion of a single new edge, not multiple edges incident to a single time-point.



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

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

(c) The resulting loop in the LO-graph

Figure 13: The RUL$^-$ algorithm's processing of the sample STNU from Figure 3

time-point at the top of S; otherwise, some element of $\mathcal{U}$ is pushed onto S—while remaining in $\mathcal{U}$ (Lines 18–19).

The algorithm performs at most $2k$ iterations because the `if` case (Lines 12–14) can be executed at most k times without encountering a negative cycle; and the `else` case (Lines 15–19) can be executed at most k times because there are only k time-points that can be popped from $\mathcal{U}$. (Once removed, no time-points are ever pushed back onto $\mathcal{U}$.) The complexity of the algorithm is therefore the complexity of Bellman-Ford (Lines 1–2) followed by running Dijkstra (in CloseRelaxLower and UpdatePotential) at most $4k$ times on a graph containing at most $m + kn$ edges, which reduces to: $O(mn + k^2n + kn\log n)$.

Figure 13 illustrates how the RUL$^-$ algorithm discovers that the sample STNU from Figure 3 is not DC. First, processing the UC-edge $(C_1, C_1{:}{-3}, A_1)$ generates the edges $(C_2, 5, A_1)$ and $(X, 9, A_1)$, shown as blue dashed edges in Figure 13a. Next, processing the UC-edge $(C_2, C_2{:}{-10}, A_2)$ generates the edge $(X, -1, A_2)$, shown as red dashed edge in Figure 13b. With these new edges, there is now a negative loop in the LO-graph, as shown in Figure 13c, which implies that the next attempt to update the potential function will fail, signaling that the STNU must be non-DC.

Interestingly, if the edge from X to C_1 were 11 instead of 12, then the

dotted edge from X to C_2 in Figure 13b would have length 8 instead of 9. Since $8 < \Delta_{C_2}$ and the only edge entering X leads back to C_1, which has already been processed by that time, no further back-propagation from X is available. However, the non-length-preserving case of the Upper$^-$ rule would still generate the edge $(X, -1, A_2)$, leading to the same conclusion.

3 A New Approach to the RUL$^-$ Algorithm

This section introduces a new approach to the RUL$^-$ algorithm that achieves an order of magnitude improvement in performance over the original algorithm. Although it makes several important modifications, the new algorithm preserves the following common features from the original algorithm:

- The focus is on reducing away upper-case edges by back-propagating along paths in the LO-graph.
- The back-propagation uses Dijkstra's algorithm, guided by a potential function that is first computed by an initial call to the Bellman-Ford algorithm, and then periodically updated by the UpdatePotential function (cf. Algorithm 8).

The most important differences include:

- It significantly reduces the number of edges that are inserted into the STNU graph while processing UC-edges, thereby significantly reducing the amount of constraint propagation required by the many instances of Dijkstra's algorithm. In particular, the new algorithm:
  - only inserts new edges generated by the *length-preserving* case of the Upper$^-$ rule, which effectively reduce away upper-case edges;
  - only *accumulates, but does not insert* any edges terminating at contingent time-points (i.e., edges generated by the Relax$^-$ or Lower$^-$ rules); and
  - *neither accumulates nor inserts* any edges generated by the *non-length-preserving* case of the Upper$^-$ rule.

  For example, whereas the original RUL$^-$ algorithm would compute *and insert* all of the ten dotted and dashed edges shown in Figure 11, the new algorithm (1) only computes, but does not insert the five dashed edges terminating at C, and (2) does not even compute the four edges of length -8 terminating at A. As will be seen, *not* inserting the edges generated by the non-length-preserving case of the Upper$^-$ rule can sometimes require extra calls to Dijkstra to ascertain whether certain *non-negative* loops in the LO-graph may, despite their non-negative length, cause the STNU to be non-DC. However, our extensive

empirical evaluation demonstrates that this cost is more than offset by the overwhelming benefit of reducing the total number of edges inserted into the STNU 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 original algorithm. In contrast, when the interrupted processing of an upper-case edge E resumes, the original algorithm begins its reprocessing of E *from scratch* (Lines 7-10 of Algorithm 7) because the intervening processing of other upper-case edges typically results in new edges being inserted into the graph and, hence, modifies the potential function.

- When processing a given 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 calls to Dijkstra to update the potential function. In contrast, in the original algorithm, the CloseRelaxLower and ApplyUpper functions (Algorithm 9) insert new edges into the graph, and the potential function is updated *before* checking whether any interruptions may be needed (cf. Lines 8-12, Algorithm 7).

The rest of this section describes the novel features of the new RUL algorithm in more detail.

The non-length-preserving case of the Upper$^-$ rule. For convenience, let $\text{Upper}^-_{\text{nlp}}$ denote the *non-length-preserving* case of the Upper^- rule (i.e., the case where $v_i - y_i < -x_i$ in Table 4). A careful review of the proof of correctness for the original RUL$^-$ algorithm (Cairo et al., 2018) reveals that the $\text{Upper}^-_{\text{nlp}}$ rule is only used for two purposes:

(1) To prove that a cycle of interrupted processings of UC-edges necessarily implies that the original STNU is not DC.

(2) To deal with the case where back-propagation from a UC-edge $(C, C{:}{-y}, A)$ encounters a *non-negative* loop $\mathcal{L}$ in the LO-graph from C back to C, where $0 \leq |\mathcal{L}| < \Delta_C$.

Case (1) is illustrated in Figure 14 where, for convenience, it is assumed that the lower bound on each contingent link is 1, and hence that each edge generated by the $\text{Upper}^-_{\text{nlp}}$ rule has length -1. Without loss of generality,

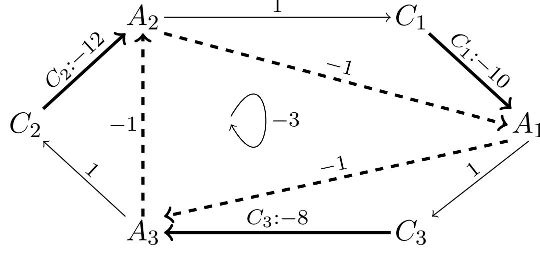
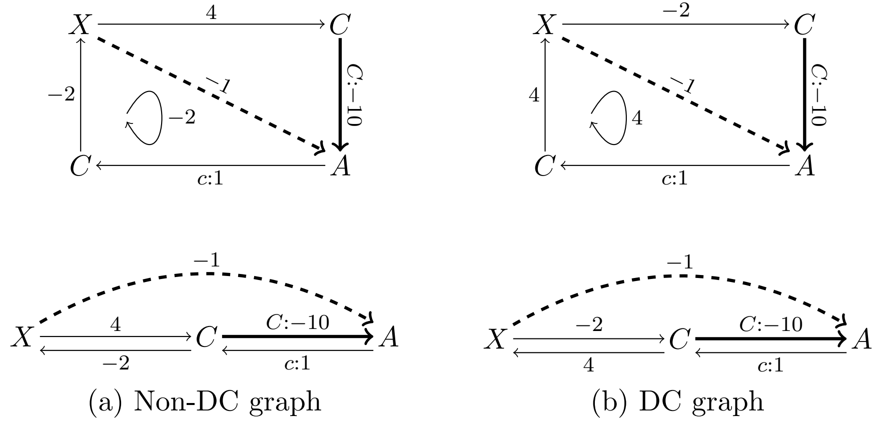


Figure 14: Using the $\text{Upper}^-_{\text{nlp}}$ rule to confirm that a cycle of interruptions ensures that the STNU is not DC



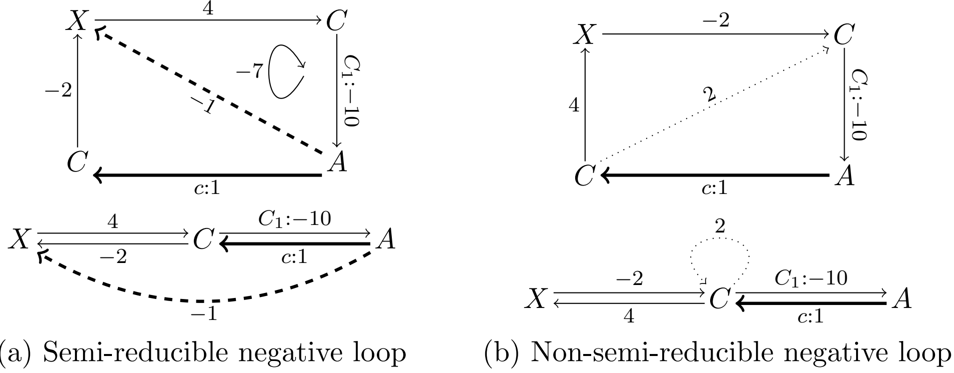
(a) Non-DC graph

(b) DC graph

Figure 15: Two scenarios in which back-propagation from the UC-edge $(C, C{:}{-10}, A)$ encounters a non-negative loop $\mathcal{L}$ from C back to C such that $|\mathcal{L}| = 2 < 9 = \Delta_C$

suppose that the UC-edge $\mathbf{E_1} = (C_1, C_1{:}{-10}, A_1)$ is processed first. Then the $\text{Upper}^-_{\text{nlp}}$ rule generates the dashed edge $(A_2, -1, A_1)$, whereupon further processing of $\mathbf{E_1}$ is interrupted by the UC-edge $\mathbf{E_2} = (C_2, C_2{:}{-12}, A_2)$. Processing $\mathbf{E_2}$ similarly begins by applying the $\text{Upper}^-_{\text{nlp}}$ rule to generate the dashed edge $(A_3, -1, A_2)$, whereupon further processing of $\mathbf{E_2}$ is interrupted by the UC-edge $\mathbf{E_3} = (C_3, C_3{:}{-8}, A_3)$. Processing $\mathbf{E_3}$ similarly uses the $\text{Upper}^-_{\text{nlp}}$ rule to generate the dashed edge $(A_1, -1, A_3)$, which completes a negative loop in the LO-graph, implying that the original STNU graph must be non-DC. Although this reasoning is perfectly correct, the algorithm does not need to duplicate its every detail by actually inserting the dashed edges into the graph. Instead, it can use the technique employed by Morris' 2014 DC-checking algorithm which involves simply monitoring for the presence of a cycle of recursive calls and, if such a cycle is ever found, immediately concluding that the network is not DC.

Case (2) is illustrated by the two contrasting scenarios shown in Figure 15. Each scenario is illustrated in two ways: the top picture includes two copies



(a) Semi-reducible negative loop (b) Non-semi-reducible negative loop

Figure 16: How Morris' 2006 DC-checking algorithm processes the scenarios from Figure 15

of the contingent time-point C to help clarify the loop being considered, whereas the bottom picture includes only one copy of each time-point. In both scenarios, processing the UC-edge $(C, C{:}{-10}, A)$ involves back-propagating along a two-edge loop $\mathcal{L}$ from C to X to C, where $0 \leq |\mathcal{L}| = 2 < 9 = \Delta_C$; however, in the lefthand scenario $\mathcal{L} = (C, -2, X, 4, C)$, whereas in the righthand scenario $\mathcal{L} = (C, 4, X, -2, C)$. This slight difference causes the lefthand graph to be non-DC, while the righthand graph is DC.

In particular, in the lefthand graph, the upper-case edge $(C, C{:}{-10}, A)$ is immediately reduced away by an application of the $\text{UPPER}^-_{\text{nlp}}$ rule to the edges $(X, 4, C)$ and $(C, C{:}{-10}, A)$, generating the dashed edge $(X, -1, A)$. This creates a negative cycle $(A, c{:}1, C, -2, X, -1, A)$ in the LO-graph, which the RUL$^-$ algorithm will detect the next time it tries to update the potential function.[16] The key features in this instance are that the lengths of the lower-case 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 similarly generates the dashed edge $(X, -1, A)$, no negative loop in the LO-graph arises because, in this scenario, the weight on the edge $(C, 4, X)$ is non-negative. In particular, the loop $(A, c{:}1, C, 4, X, -1, A)$ has length $4 \geq 0$.

Figure 16 illustrates how Morris' 2006 DC-checking algorithm would handle the two scenarios from Figure 15. In the lefthand graph, the lower-case edge $(A, c{:}1, C)$ would be immediately reduced away by an application of the Lower-Case rule to the edges $(A, c{:}1, C)$ and $(C, -2, X)$, generating the dashed edge $(A, -1, X)$, thereby creating a negative loop $(A, -1, X, 4, C, C{:}{-10}, A)$ in the OU-graph, which would be detected the next time the potential function was computed. In contrast, in the righthand

[16]Both scenarios would also generate the edges, $(C, 2, C)$ and $(C, -1, A)$, but they are irrelevant to the current discussion.

graph, the lower-case edge cannot be reduced away. In particular, forward propagation from C along the path $(C, 4, X, -2, C)$ would generate the dotted edge $(C, 2, C)$ which, being a non-negative loop, would halt any further propagation. In addition, forward propagation from C could not make use of the upper-case edge $(C, C{:}{-10}, A)$, since the Cross-Case rule does not allow the combination of lower-case and upper-case edges associated with the same contingent link. As a result, Morris' 2006 algorithm would detect the semi-reducible negative loop in the lefthand graph and declare it to be non-DC but, finding no such SRN loop in the righthand graph, would declare it to be DC.

Whereas the original RUL$^-$ algorithm uses the $\text{Upper}^-_{\text{nlp}}$ rule to distinguish the contrasting scenarios from Figure 15, the new algorithm introduced below is able to distinguish them without using the $\text{Upper}^-_{\text{nlp}}$ rule. In particular, when back-propagating from C as part of processing the UC-edge $(C, C{:}{-y}, A)$, the new algorithm keeps track of whether it ever encounters a loop from C back to C whose length is less than Δ_C. If so, it would, at the appropriate time, carry out a separate forward propagation from C in the OU-graph that is similar to the forward propagation from Morris' 2006 algorithm, except that it would restrict attention to those nodes X that were encountered during the original back-propagation from C along a shortest path of length less than Δ_C (i.e., those X for which the $\text{Upper}^-_{\text{nlp}}$ rule would apply if it were being used).[17] If, during that separate forward propagation, the new algorithm were to discover that it was able to reduce away the lower-case edge $(A, c{:}x, C)$, then it would immediately stop all processing and conclude that the network was not DC; otherwise, it would resume normal processing.

Figure 17 illustrates how the new algorithm would deal with the scenarios from Figure 15. In each case, processing the upper-case edge $(C, C{:}{-10}, A)$ would generate (but not insert) the dotted loop from C to C whose length is $2 < 9 = \Delta_C$. Therefore, in each case, the algorithm would perform a separate forward propagation from C in the OU-graph looking for opportunities to reduce away the lower-case edge $(A, c{:}1, C)$, but only exploring nodes X that were encountered during the original back-propagation from C. As already seen in Figure 16, the lower-case edge in the lefthand graph is able to be reduced away, but the lower-case edge in the righthand graph is not. Therefore, the new algorithm would immediately declare the lefthand graph to be non-DC, but would resume normal processing for the righthand graph—assuming there were other upper-case edges to be explored.

Next, recall how the RUL$^-$ algorithm processed the sample STNU from Figure 3, as seen previously in Figure 13. The new DC-checking algorithm

[17] Recall that if back-propagation encountered a shortest path from C back to X of some length greater than or equal to Δ_C, then the back-propagation would stop at X. Therefore, the only way X could be an *interior* point along a shortest path from C back to C would be if the shortest path-length from X to C was less than Δ_C.

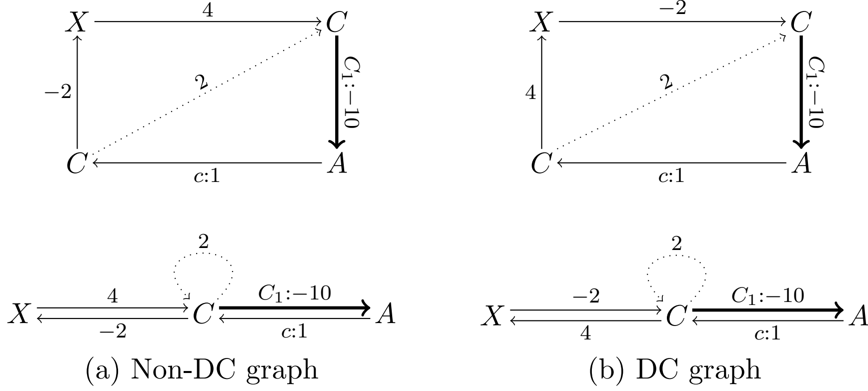
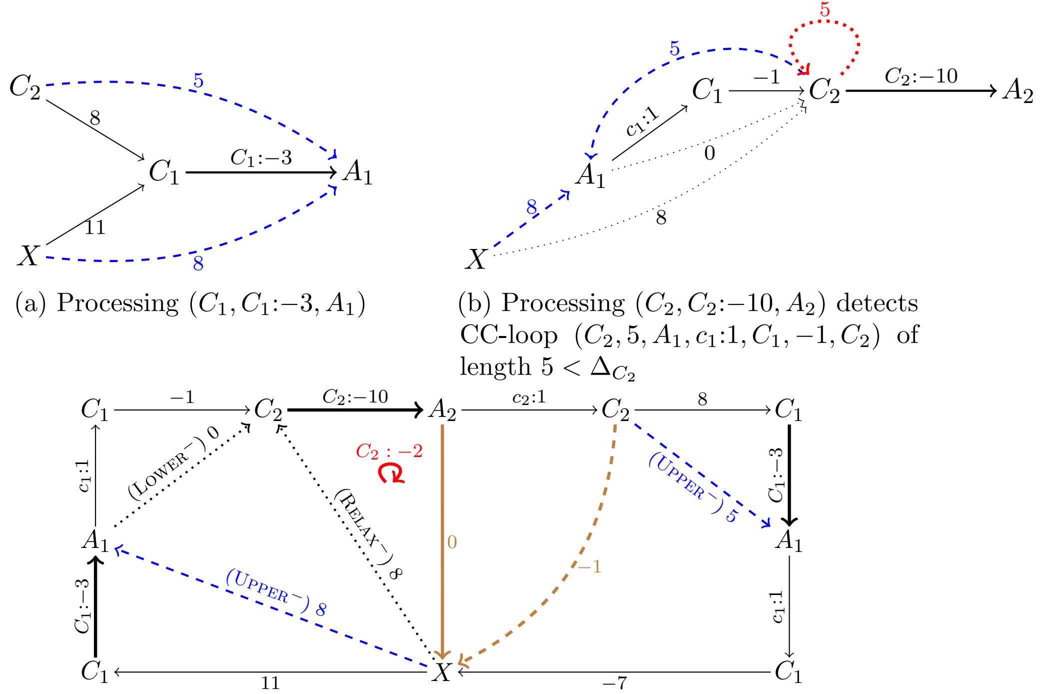


Figure 17: How the new RUL algorithm processes the scenarios from Figure 15 without using the $\text{UPPER}^{-}_{\text{nlp}}$ rule

would process the sample STNU from Figure 3 in the same way, since that example does not require the non-length-preserving case of the UPPER^{-} rule. However, if the edge from X to C_1 in the sample STNU were 11 instead of 12, then, as discussed previously, the RUL^{-} algorithm would require the non-length-preserving case of the UPPER^{-} rule, which is not available to the new algorithm.

Figure 18 shows how the new algorithm would process this slightly tighter version of the sample STNU. In particular, as shown in Figure 18b, back-propagation from C_2 detects a loop from C_2 back to C_2 of length $5 < \Delta_{C_2}$. In turn, this triggers a separate forward propagation from C_2 which, as shown in Figure 18c, detects an extension sub-path (dashed, brown) for the LC-edge $(A_2, c_2{:}1, C_2)$. Reducing away the LC-edge generates an edge (brown) from C_2 to X whose length is necessarily less than 1 (i.e., less than the lower bound for the contingent link $(A_2, 1, 10, C_2)$). Since the forward propagation only visited edges in the LO-graph involving time-points for which the back-propagation in Figure 18b found paths to C_2 of length less than Δ_{C_2}, the edge from A_2 to X, together with the path from X to C_2 of length $8 < \Delta_{C_2}$ discovered during back propagation, ensures that the path from A_2 to X to C_2 to A_2 will be a negative loop in the OU-graph (shown in red) and, hence, that the network must be non-DC.

Although the technique of avoiding the non-length-preserving case of the UPPER^{-} rule occasionally involves doing extra forward propagations guided by Dijkstra, the cost of doing so is expected to be very small. First, this extra processing never inserts any new edges since, if the relevant lower-case edge can be reduced away, the algorithm immediately halts; therefore, no extra updating of potential functions is ever required. Second, discovering such loops from C back to C during back-propagation from an upper-case



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

(b) Processing $(C_2, C_2{:}{-10}, A_2)$ detects 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 to A_1 to C_1 to X detects an extension sub-path (dashed, brown) for the LC-edge $(A_2, c_2{:}1, C_2)$ enabling it to be reduced away (solid, brown), yielding a negative loop in the OU-graph (red), implying that the STNU is not DC

Figure 18: The new DC-checking algorithm's processing of a tighter version of the sample STNU from Figure 3, where the edge from X to C_1 has length 11, not 12

edge $(C, C{:}{-y}, A)$ is expected to be quite rare. Finally, the cost of the extra forward traversals is expected to be far outweighed by the benefit of *not* using the $\text{Upper}^-_{\text{nlp}}$ rule and, hence, *not* inserting all of the extra edges that it would generate.

3.1 Pseudocode for the New RUL DC-Checking Algorithm

The high-level structure of the new RUL DC-checking algorithm is given as Algorithm 10. It creates a `globalInfo` data structure (Line 1) that will be passed to various helper functions. The `globalInfo` data structure holds a potential function that is initialized at Line 2 by a call to the Bellman-Ford algorithm (Algorithm 8), and a status vector that holds the processing status of each of the k UC-edges. Initially, the status of each UC-edge is set to `unstarted` (Line 4). Then, mirroring the structure of Morris' 2014 algorithm (Algorithm 5), it calls the recursive helper function `recRULbackprop` on each of the k UC-edges.

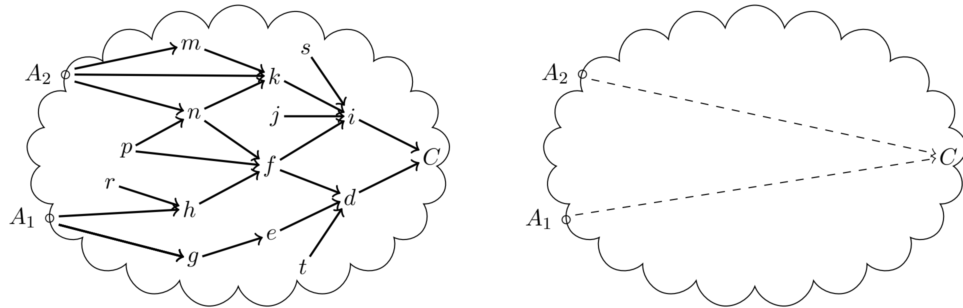
Pseudocode for the recursive helper function `recRULbackprop` is given as Algorithm 11. Similarly to Morris' DCbackprop function (Algorithm 6), the `recRULbackprop` function begins (Lines 3–4) by checking the status of the input UC-edge $\mathbf{E} = (C, C{:}{-y}, A)$. A status of `started` signals a cycle of recursive interruptions, implying that the network is not DC. A status of `finished` indicates that this UC-edge has already been processed and hence can be ignored. The rest of the function handles the case where the UC-edge has not yet been processed.

Line 7 initializes a `localInfo` data structure that has three fields that will be updated during the subsequent back-propagation from C: (1) `CC_loop?`, a boolean flag (initially false) that tracks whether back-propagating has discovered a non-negative loop from C back to C, as discussed previously in Figure 15; (2) `unstarted-UCEs`, a list (initially empty) of UC-edges that have been encountered during back-propagation with a status of `unstarted`, each of which would signal the need to interrupt the processing of $\mathbf{E}$; and (3) `distFrom`, a vector that specifies for each node X the shortest distance (initially ∞) from X to C that has been discovered so far while back-propagating along LO-edges from C.

The back-propagation from C along shortest paths in the LO-graph is guided by a priority queue $\mathcal{Q}$ that initially contains all nodes X for which there is an *ordinary* edge (X, δ_{XC}, C) terminating at C (Lines 11–12). This is the same queue initialization that the RUL$^-$ algorithm uses in the CloseRelaxLower function (Algorithm 9). Lower-case edges are ignored at the first step of back-propagation because the lefthand edge in the Upper$^-$ rule (Table 4) must be ordinary. And, as in the CloseRelaxLower function, the key for each X inserted into the queue is $h(X) + \delta_{XC}$ instead of the typical $h(X) + \delta_{XC} - h(C)$ because the terminus of all such distances is fixed at C, and therefore it is not necessary to include the common term $-h(C)$.

The `while` loop at Lines 14 to 23 performs the back-propagation from C along edges in the LO-graph, but organizes the propagation much differently than the original RUL$^-$ algorithm. Recall that the RUL-DC-check function (Algorithm 7) from the RUL$^-$ algorithm processes $\mathbf{E}$ by propagating back from C, and then generating and inserting all edges found by the Relax$^-$, Lower$^-$ and Upper$^-$ rules *before* checking whether any `unstarted` UC-edges were encountered (i.e., whether any interruptions are necessary). If any such UC-edges were encountered, it suspends its processing of $\mathbf{E}$ to process one of the interrupting UC-edges. Once that UC-edge has finished processing, it returns to $\mathbf{E}$, starting from scratch (because an interruption typically changes the potential function), redoing the entire back-propagation, again checking for any unstarted UC-edges, processing an interruption, then returning to process $\mathbf{E}$ again from scratch, and so on. Only once all interruptions from $\mathbf{E}$ have been processed, can the processing of $\mathbf{E}$ eventually conclude.

The new algorithm avoids the kinds of redundant computations described above by dividing the back-propagation into rounds. Each round begins



(a) A cloud of back-propagation from C that encounters activation time-points A_1 and A_2 for two `unstarted` UC-edges

(b) Initialization for the next round of back-propagation from C

Figure 19: Initial back-propagation from C encountering interruptions, and then re-starting

(at Line 15) by calling the `OneStepBackProp` function (Algorithm 12). The `OneStepBackProp` function uses the Relax$^-$ and Lower$^-$ rules to back-propagate from C along LO-edges, but it does *not* insert any edges into the graph. Instead, it merely accumulates information in the `localInfo` structure for later use. In particular, it stores the lengths of the shortest paths it discovers in the relevant slots of the `distFrom` vector (Line 9); it sets the `CC_loop?` flag if it ever discovers a loop from C to C of length less than Δ_C (Line 13); and it accumulates any encountered UC-edges whose status is `unstarted` in the `unstarted-UCEs` field. (Further detail about `OneStepBackProp` will be given later.)

When `OneStepBackProp` returns, the `recRULbackprop` function checks whether any unstarted UC-edges were encountered (Line 16). If so, then each such UC-edge is recursively processed by `recRULbackprop` (Lines 17 to 18), and the queue $\mathcal{Q}$ is cleared and re-initialized in preparation for the next round (Lines 19 to 22). Although the intervening interruptions typically cause the potential function to be updated, it is not necessary to restart the processing of **E** from scratch. Instead, the back-propagation in the next round can resume from where it left off: namely, from the activation time-points of the interrupting UC-edges. The basic idea is illustrated in Figure 19. The lefthand image shows a “cloud” of back-propagation from C done in the previous round, which recursively applies the Lower$^-$ and Relax$^-$ rules as much as possible, except that it does not propagate past activation time-points for `unstarted` UC-edges, called A_1 and A_2 in the figure. For the next round, it is not necessary to re-do the propagations in this “cloud”, since the distance information is stored in the `distFrom` vector. It is only necessary to initialize the queue with the activation time-points for each `unstarted` UC-edge that was encountered in the previous round, using the

updated potential function to compute the adjusted distance. In particular, each such activation time-point X is inserted into the queue with a key of (`localInfo.distFrom`[X] + `globalInfo.potFunc`[X]) (Line 20). (As before, the common term $-h(C)$ is left out.) After doing so, the `distFrom` entry for each of these activation time-points is set to ∞ (Line 22) to ensure that they are handled properly by the next call to the `OneStepBackProp` function. (See the `if` expression at Line 8 of `OneStepBackProp`.) All other `distFrom` entries are preserved because they hold the lengths of shortest paths found in prior rounds. (Note that `distFrom` entries hold actual path lengths, not adjusted path lengths. This facilitates their conversion to adjusted lengths based on *any* potential function.)

After the queue has been re-initialized, another round of back-propagation is conducted by `OneStepBackProp`. For each activation time-point A_i that was used to initialize the queue, back-propagation will commence along pre-existing LO-edges terminating at A_i, as well as any new (ordinary) edges that were generated from the processing of the UC-edge associated with A_i. This round of back-propagation may lead to the discovery of shorter paths to nodes that were visited in prior rounds (i.e., that were part of the "cloud" in Figure 19), as well as paths to nodes that had not yet been visited.

Managing loops from C back to C discovered during back-propagation. Once all rounds of back-propagation from C have been completed (i.e., once the `while` loop spanning Lines 14 to 23 in Algorithm 11 terminates), the `recRULbackprop` function checks whether a loop from C back to C was ever discovered (Line 24) and, if so, (at Line 25) calls the `fwdPropNotDC` function (Algorithm 13) to do a restricted kind of *forward* propagation, as discussed previously with Figures 15 to 17. Because the back-propagation that discovered the loop from C back to C only traversed edges in the LO-graph and only propagated back from nodes X for which `distFrom`[X] $< \Delta_C$ (recall Line 10 in `OneStepBackProp` (Algorithm 12)) the forward propagation done by the `fwdPropNotDC` function propagates forward from the contingent time-point C only along LO-edges, and only propagating forward from nodes X for which `distFrom`$(X, C) < \Delta_C$. The goal of the forward propagation is to find a negative-length path that could be used to reduce-away the LC-edge $(A, c{:}x, C)$.

As usual, to enable Dijkstra to guide the traversal of shortest paths, all distances from C to X discovered during the forward propagation are converted into non-negative values, as follows: $\mathit{dist}(C, X) \mapsto h(C) + \mathit{dist}(C, X) - h(X)$, where h is a potential function for the LO-graph. However, since each path starts at C, the common term $h(C)$ can be ignored.

The priority queue $\mathcal{Q}$ is initialized to include only the starting point C, with a key of $\mathit{dist}(C, C) - h(C) = -h(C)$ (Lines 1–2). The `while` loop (Lines 3–10) implements the forward propagation. Only nodes X for which

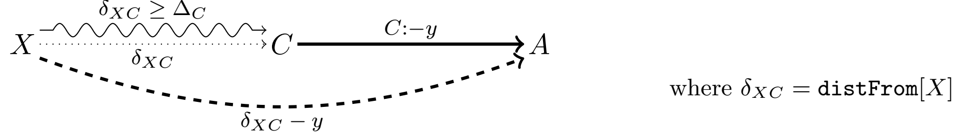


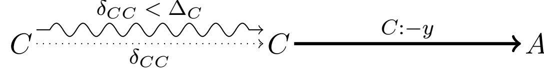
Figure 20: Reducing away a UC-edge using the *length-preserving* case of UPPER$^-$

`distFrom`$[X] < \Delta_C$ are considered (Line 6). If a node X is ever found for which $dist(C, X) < 0$, that signals the existence of a path in the LO-graph that can be used to reduce-away the LC-edge $(A, c{:}x, C)$, which implies that the network must be non-DC (Line 7). Otherwise, forward propagation from X along each LO-edge (X, δ_{XY}, Y) emanating from X is used to insert Y into the queue or decrease its key, as appropriate (Lines 8–10). If the forward propagation does not find a way to reduce-away the LC-edge, then the `fwdPropNotDC` function returns $\perp$ (Line 11).

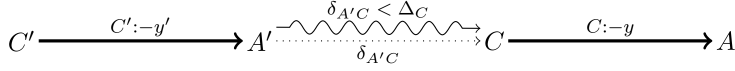
If `fwdPropNotDC` fails to find a way to reduce-away the LC-edge, then `recRULbackprop` (finally) applies the *length-preserving* case of the UPPER$^-$ rule, as illustrated in Figure 20, to generate new edges and *insert* them into the STNU graph (Lines 27–32). Note that this rule is only applied if $\Delta_C \leq$ `distFrom`$[X] < \infty$ (Line 30). In addition, it is important to highlight that these are the *only* new edges that are inserted into the STNU graph as a result of processing the UC-edge $\mathbf{E}_X$, and these insertions are done only *after* all interrupting UC-edges were processed (Lines 17–18). If the UPPER$^-$ rule does generate one or more new edges that need to be inserted into the graph, then the potential function is updated (Lines 34–35), using the same (improved) UpdatePotential function as in the RUL$^-$ algorithm (Algorithm 8). Note that, if needed, this is the only time that the potential function is updated during the processing of the UC-edge $\mathbf{E}_X$. After that, the UC-edge has been fully processed, so its status is set to `finished` (Line 36), and the `recRULbackprop` function returns $\top$ (Line 37).

Further details of the `OneStepBackProp` function. The back-propagation from C along LO-edges done by the `OneStepBackProp` function effectively replaces the work done by CloseRelaxLower in the original RUL$^-$ algorithm. However, it does not insert new edges; instead, it only accumulates distance information, keeps track of whether a loop from C back to C has been found, and accumulates `unstarted` UC-edges whose activation time-points have been encountered.

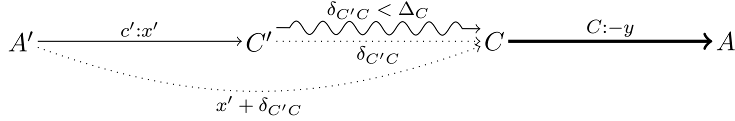
Whenever a time-point X is popped from the queue (Line 5), the discovered distance, δ_{XC} from X to C is retrieved from the key using the conversion at Line 6. Since propagations in a prior round may have found a shorter



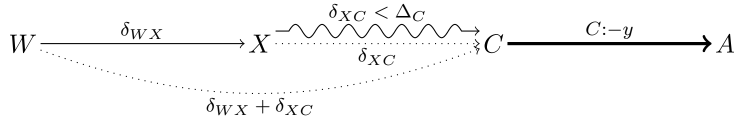
Case 1 ($X \equiv C$): Encountering a loop from C back to C



Cases 2 & 3 ($X \equiv A'$): Meeting a UC-edge with status `unstarted` or `started`



Case 4a ($X \equiv C'$): Back-propagation across an LC-edge using the LOWER$^-$ rule



Case 4b: Back-propagation across an ordinary edge using the RELAX$^-$ rule

Figure 21: Different cases of back-propagation in `OneStepBackProp` (Algorithm 12)

distance—recall the "cloud" in Figure 19—processing of X only continues if δ_{XC} is indeed smaller than the distance from X to C found in a prior round (Line 8).[18] And since the LOWER$^-$ and RELAX$^-$ rules only apply when the righthand edge has a weight less than Δ_C, back-propagation from X only continues if $\delta_{XC} < \Delta_C$ (Line 10). The rest of the code considers the following cases, each illustrated in Figure 21:

(Case 1) $X \equiv C$ (Line 11). In this case, a loop from C back to C has been discovered. If the length δ_{XC} is negative, then the network must be non-DC (Line 12). Otherwise, the `CC_loop?` flag is set (Line 13), which will (eventually) trigger a separate forward propagation by `fwdPropNotDC`, as described earlier. Since the original back-propagation started from C, no further back-propagation from $X \equiv C$ is done in this case.

(Case 2) X is an activation time-point for an `unstarted` UC-edge $\mathbf{E}_X$.[19]

[18] The reason `distFrom`[X] was set to ∞ for each activation time-point initially inserted into the queue at Line 22 of `recRULbackprop` (Algorithm 11) was precisely so that the inequality at Line 8 of `OneStepBackProp` would succeed, enabling those time-points to be processed in the current round.

[19] At Line 7, the `getUCEdgeFromATP` method is used to fetch the UC-edge associated

In this case, the pair, $(\mathbf{E}_X, X)$ is added to `unstarted-UCEs` and no further back-propagation is done from X.

(Case 3) X is an activation time-point for an `unfinished` UC-edge (Lines 16–17), signaling a cycle of recursive calls, which implies that the network is not DC; hence the algorithm immediately returns $\perp$.

(Case 4) In this case, one of the following holds: (1) X is an activation time-point for a `finished` UC-edge; (2) X is a contingent time-point C' other than C; or (3) X is neither an activation nor a contingent time-point. As a result, back-propagation from X can continue using the Lower$^-$ and Relax$^-$ rules (Lines 18–22). The back-propagation is handled by the `NewApplyRelaxLower` function (Algorithm 14), which is identical to the ApplyRelaxLower function (Algorithm 9) from the original RUL$^-$ algorithm, except that it allows back-propagating to $W \equiv C$ (to enable catching the loops handled by Case 1), whereas `ApplyRelaxLower` explicitly rules it out (cf. the constraint $V \not\equiv C$ in Line 5 of `NewApplyRelaxLower`).

Summary of the new RUL DC-checking algorithm. The new RUL DC-checking algorithm presented in this section aims to improve the performance of DC checking 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 Lower$^-$ and Relax$^-$ rules while refraining from inserting any new edges associated with those paths and, second, only using the length-preserving case of the Upper$^-$ rule. Although avoiding the non-length-preserving case of the Upper$^-$ rule requires occasionally performing separate forward propagations, it is expected that the savings from not inserting so many edges will far outweigh the cost of those 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. The next section reports on a thorough empirical evaluation of the new algorithm on a wide variety of benchmark problems and demonstrates that the new algorithm achieves an order-of-magnitude improvement over existing DC-checking algorithms for STNUs.

---

with X if X happens to be an activation time-point. If X is not an activation time-point, then $\mathbf{E}_X = \perp$. Note that this assumes that each contingent link has a unique activation time-point, which is not necessarily the case for every STNU. Therefore, as in Morris' 2014 algorithm, a simple linear-time pre-process can be used to convert the input STNU into a network for which this property holds. Indeed, converting to normal form, as described for Morris' 2014 algorithm, would suffice.

4 Experimental Evaluation

This section compares the performance of our new RUL DC-checking algorithm against the pre-existing RUL^- and Morris 2014 algorithms. RUL2020 refers to our implementation of Algorithm 10, while RUL^- refers to our implementation of Algorithm 7 and Morris2014 our implementation of Algorithm 5. All algorithms and procedures were implemented in Java and executed on a JVM 8 having 8GB of heap memory on a Linux box with one Intel(R) Xeon(R) CPU E5-2637 v4 @ 3.50GHz. The implementations of all algorithms and procedures are freely available as a Java Package (Posenato, 2020)

We tested each implementation on random instances obtained from an STNU generator that we set up for this evaluation. The STNU generator can build random instances having a chosen topology that can be tuned by a variety of input parameters. The possible topologies are *no-topology, tree*, and *worker-lanes*. After some testing, we verified that the *worker-lanes* topology, which simulates the worker-lanes of business process modeling (Object Management Group (OMG), 2007), is the most interesting because it allows the generation of random instances where there could be circuits involving many constraints. In this topology, the set of contingent links is partitioned into a specified number of lanes, the contingent links in each lane representing a sequence of tasks that must be executed by some agent. In addition, the contingent links within each lane are interspersed with ordinary constraints that specify delays between the end of one task and the start of the next. Finally, the random generator inserts extra constraints between pairs of nodes that belong to different lanes to represent temporal-coordination constraints among tasks executed by different agents. Typically, such constraints involve nodes on different lanes that are at a similar distance from the start of their respective lanes. As an example, Figure 22 depicts a portion of a random STNU having 500 nodes and 50 contingent links in a 5-worker-lane topology.

Many aspects of the worker-lanes topology can be tuned as input parameters to the generator (e.g., the number of nodes, the number of contingent links, the number of lanes, the probability of a temporal constraint for a pair of nodes from different lanes, the maximum weight of each contingent link, the maximum weight of each ordinary constraint, and so on).

Test 1. For our first evaluation, called Test 1, we generated instances using the following parameters:

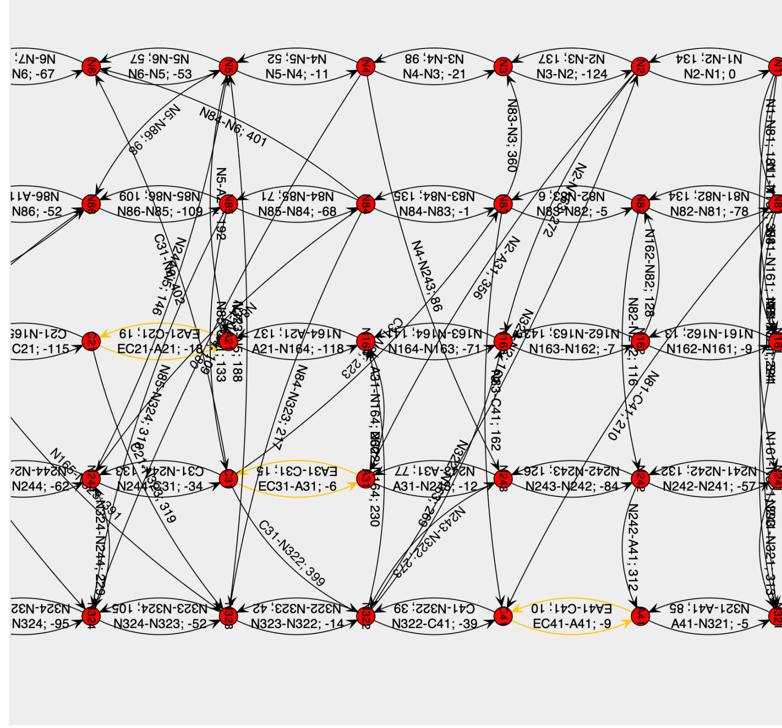


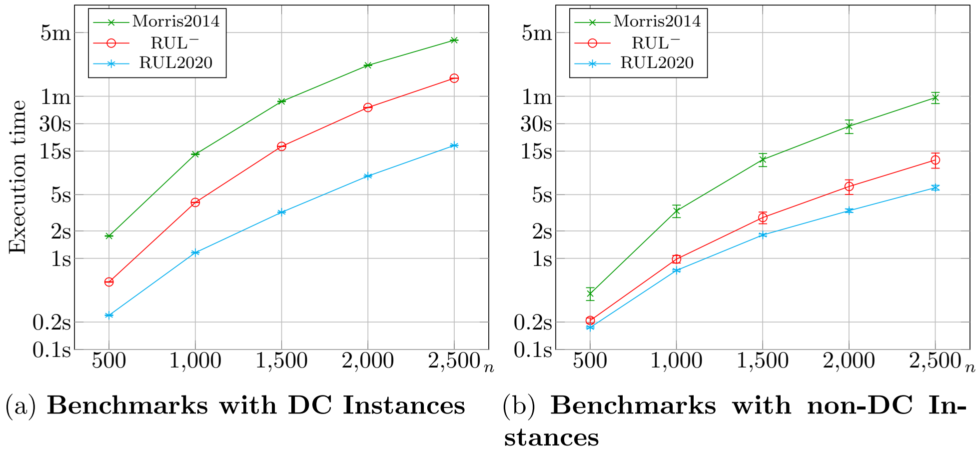
Figure 22: An example of a randomly generated STNU

Number of nodes, n	$n \in \{500, 1000, 1500, 2000, 2500\}$
Number of lanes	5
Number of contingent links, k	$k = n/10$ (hence, $k = O(n)$)
Max absolute weight of ordinary edges	150
Max contingent range	$[0, 20]$
Probability of constraint among nodes in different lanes	0.40

For these parameter choices, it follows that each node has two incoming edges and two outgoing edges in the same lane, as well as an average of 2.56 incident edges (for non-activation time-points) representing temporal-coordination constraints with nodes in other lanes.[20] Therefore, the number of edges is, on average, $6.56n - 2.56k - 10$; hence, $m = O(n)$. For each value of $n \in \{500, 1000, 1500, 2000, 2500\}$, we generated 200 DC networks and 200 non-DC networks. Thus, there are ten sub-benchmarks, each containing 200 instances.

Figure 23 displays the average execution times of the three algorithms across all ten sub-benchmarks. Each plotted point represents the average

[20]Temporal-coordination constraints are set in a way that avoids introducing negative circuits among a pair of nodes.



(a) **Benchmarks with DC Instances** (b) **Benchmarks with non-DC Instances**

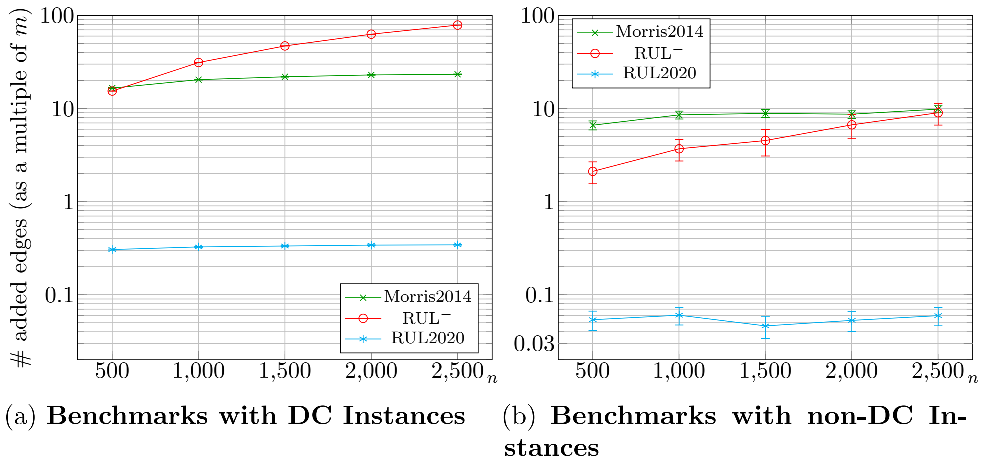
Each plotted point represents average execution time over 200 instances

Figure 23: Test 1: Execution time vs. number of nodes, n, where $k = O(n)$ and $m = O(n)$.

execution time for a given algorithm on the 200 instances of the given size, and the error bar for each point represents the 95% confidence interval. For example, over the 200 DC instances having $n = 2500$ time-points and $k = 250$ contingent links, the average execution time (in seconds) of the Morris2014 algorithm lies within the interval $[246.24, 248.56]$ with 95% confidence, while the average execution time of the RUL2020 algorithm lies within the interval $[17.26, 17.36]$ with 95% of confidence. These results demonstrate that the RUL2020 algorithm performs significantly better than the other two algorithms, especially over DC instances, but also over non-DC instances. For non-DC instances, the 95%-confidence intervals tend to be larger than those for the corresponding DC instances because for some non-DC instances the negative cycle can be detected immediately (e.g., by an initial run of Bellman-Ford or during the processing of the first contingent link or negative node), while others may require significant amounts of propagation.

One of our principal motivating hypotheses was that our new algorithm would be significantly faster than the RUL$^-$ algorithm because it inserts significantly fewer new edges into the input STNU graph. In particular, whereas the RUL$^-$ algorithm computes and inserts new edges arising from all three of the RUL$^-$ rules, the RUL2020 algorithm only inserts edges arising from the length-preserving case of the UPPER$^-$ rule. In addition, it only computes *but does not insert* intermediate edges resulting from applications of the LOWER$^-$ and RELAX$^-$ rules.

Figure 24 dramatically confirms that our new algorithm indeed inserts significantly fewer edges into the STNU graph. The plots show the number of edges inserted by each of the three algorithms as a multiple of the number of edges in the original graph (i.e., m), using a logarithmic scale. On DC



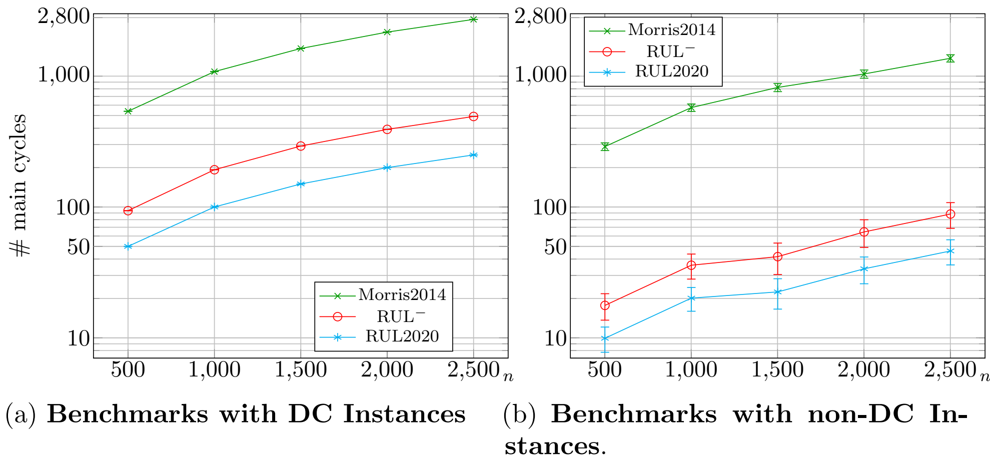
(a) **Benchmarks with DC Instances** (b) **Benchmarks with non-DC Instances**

Figure 24: Test 1: Number of added edges (as a multiple of m) vs. the number of nodes

instances, our algorithm inserted, on average, fewer than $0.4m$ new edges, while the other algorithms inserted $20m$ to $80m$ new edges. On non-DC instances, our algorithm inserted fewer than $.07m$ new edges, while the others inserted $2m$ to $7m$ new edges. Although avoiding the non-length-preserving case of the UPPER$^-$ rule sometimes requires the RUL2020 algorithm to carry out extra traversals of certain loops (recall the `fwdPropNotDC` function from Algorithm 13), the fact that our algorithm inserts so few new edges completely outweighs the minimal cost of occasional extra forward propagations.

By maintaining information about back-propagations done prior to any interruptions, the RUL2020 algorithm is able to resume processing from where it left off, after all interruptions have completed. As a result, the RUL2020 algorithm makes at most k calls to its main recursive helper function `recRULbackprop` (Algorithm 11). In contrast, the RUL$^-$ algorithm makes up to $2k$ calls to its main helper function RUL-DC-check (Algorithm 7) because it restarts its processing of an interrupted UC-edge from scratch each time. Figure 25 confirms that on DC instances, which require exhaustive processing of each UC-edge, the RUL2020 algorithm calls its main helper function k times, while the RUL$^-$ algorithm calls its main helper function $2k$ times. On non-DC instances, the pattern is similar, except that detection of a negative cycle can cause the algorithms to stop well before processing k or $2k$ UC-edges, respectively.

In sharp contrast to the RUL2020 and RUL$^-$ algorithms, the Morris2014 algorithm processes up to n negative nodes. Indeed, the conversion of the input STNU into normal form introduces k new negative nodes. For example, if $n = 2500$ and $k = n/10 = 250$, there can be up to $n + k = 2750$ negative nodes, causing the Morris2014 algorithm to make up to 2750 calls to its main recursive helper function DCbackprop (Algorithm 5). Figure 25 shows



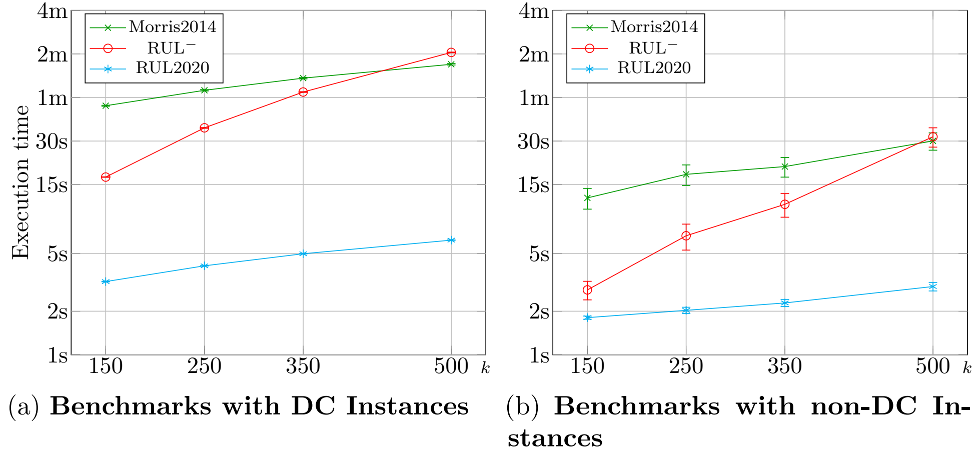
(a) **Benchmarks with DC Instances** (b) **Benchmarks with non-DC Instances**.

Figure 25: Test 1: Number of main cycles (iterations or recursive calls) vs. number of nodes

that the RUL$^-$ and RUL2020 algorithms make significantly fewer calls to their main functions (whether iterative or recursive) than the Morris2014 algorithm across the networks in our Test 1 benchmark.

Test 2. For our second test, we aimed to study the behavior of the three algorithms with respect to the number of contingent links. Therefore, we generated random instances where the number of nodes was fixed at 1500, but the number contingent links varied from 150 to 500. In particular, we built eight new benchmarks—four with 200 DC instances each, and four with 200 non-DC instances each—where $n = 1500$ and $k \in \{150, 250, 350, 500\}$. Note that when a network has 1500 nodes and 500 contingent links, the number of ordinary nodes (i.e., nodes that are neither activation nor contingent time-points) is $1500 - 2 \cdot 500 = 500$. Therefore, there is on average just one ordinary node between each consecutive pair of contingent links.

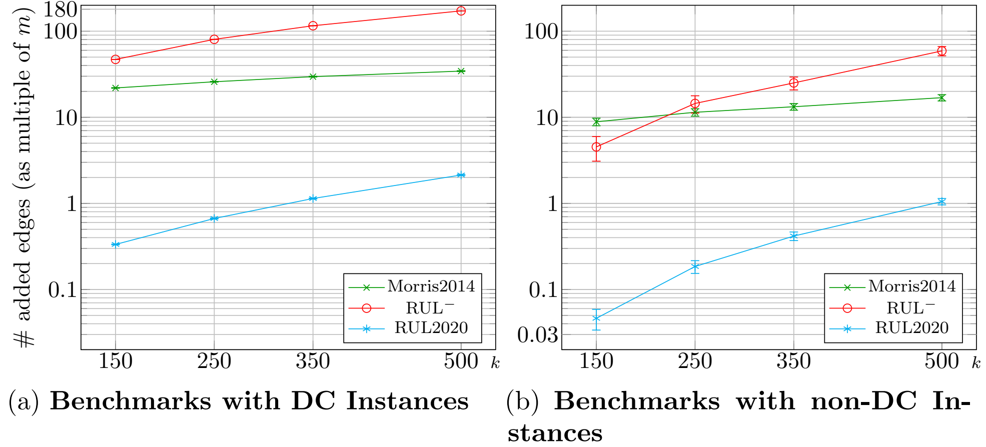
Figure 26 plots the execution times of the three algorithms across these new benchmarks. The experiment confirms that the new RUL2020 algorithm is fastest even when the number of contingent links increases. In addition, it shows that the execution time of the Morris2014 algorithm increases more slowly than the RUL$^-$ algorithm as the number of contingent links increases until, eventually, the performance of the Morris2014 algorithm becomes better than that of the RUL$^-$ algorithm when there are 500 contingent links. We believe that this occurs for two reasons: (1) just as the RUL2020 algorithm only inserts edges needed to reduce away upper-case edges, the Morris2014 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, meaning that the maximum number of times the main processing function of either algorithm is called



(a) **Benchmarks with DC Instances** (b) **Benchmarks with non-DC Instances**

Figure 26: Test 2: Execution time vs. number of contingent links in STNUs with 1500 nodes

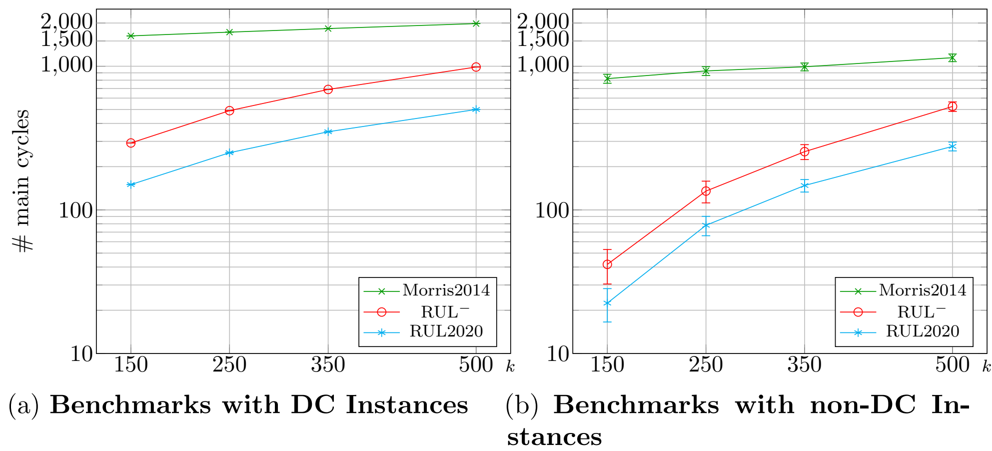
will also converge. These conjectures are supported by the plots in Figures 27 and 28.



(a) **Benchmarks with DC Instances** (b) **Benchmarks with non-DC Instances**

Figure 27: Test 2: Number of added edges (as a multiple of m) vs. the number of contingent links in STNUs with 1500 nodes

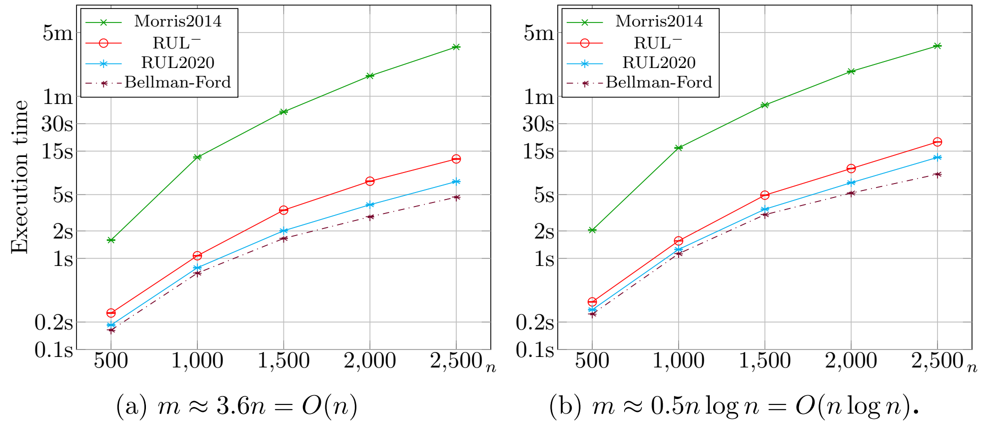
Test 3. Finally, our third test explored how the algorithms compare in very sparse networks where the number of contingent links satisfies $k \approx \sqrt{n}$. For example, in a network with 1000 nodes, k would be only 32; and in a network with 2500 nodes, k would be only 50. Figure 29 confirms that the RUL2020 algorithm once again out-performs the other two algorithms. However, the improvement over the RUL$^-$ algorithm is smaller. Indeed, most of the computation time by the two algorithms in these instances comes from



(a) **Benchmarks with DC Instances** (b) **Benchmarks with non-DC Instances**

Figure 28: Test 2: Algorithm main cycles vs number of contingent links in instances having 1500 nodes

their initial run of Bellman-Ford, which suggests that using faster versions of Bellman-Ford (e.g., as presented by Bannister and Eppstein (2012)) could help improve both of these algorithms when applied to very sparse networks.



(a) $m \approx 3.6n = O(n)$

(b) $m \approx 0.5n \log n = O(n \log n)$.

Figure 29: Test 3: Execution time vs. number of nodes in DC networks where $k \approx \sqrt{n}$.

Appendix: Proof of Correctness

The proof of correctness for the RUL^{-} algorithm in Cairo et al. (2018) overlooks an important technicality: the back-propagation from a contingent time-point C done by the algorithm need not follow the same sequence of edges that are present in the Semi-Reducible Negative (SRN) loop $\mathcal{P}$ that plays a central role in the proof. Although the basic thrust of their proof is right, correcting this oversight requires some additional work. Rather than providing a corrected version of the proof of correctness for the RUL^{-} algorithm, this section presents a proof of correctness for our new RUL2020 DC-checking algorithm.

First, Lemma 1 presents a generalization of a result by Morris (2006) that specifies conditions under which removing a sub-loop S from an SRN loop $\mathcal{P}$ preserves its semi-reducibility. Next, Lemma 2 defines three important properties of paths in STNU graphs—abbreviated as BF, CCN and NN—and shows that any non-DC STNU graph must have an SRN loop $\mathcal{P}$ for which these three properties hold. Finally, Theorem 1 states that the RUL2020 algorithm is *complete* (i.e., that if the input STNU is not DC, then the algorithm will return $\perp$).[21] The proof of Theorem 1 starts with a non-DC STNU and, by Lemma 2, an SRN loop $\mathcal{P}$ for which the BF, CCN and NN properties hold. The rest of the proof is by induction on the number of *distinct* UC-edges present in $\mathcal{P}$. The general structure of the inductive argument is shown in Figure 30. For the base case, if $\mathcal{P}$ has no UC-edges, then $\mathcal{P}$ is a negative loop in the LO-graph, which implies that the STNU is not DC. The algorithm would detect this when it tried to compute or update its potential function and, therefore, would return $\perp$. Next, suppose that

[21]The soundness of the algorithm is ensured by the soundness of the RUL^{-} propagation rules it employs, as proven by Cairo et al. (2018).

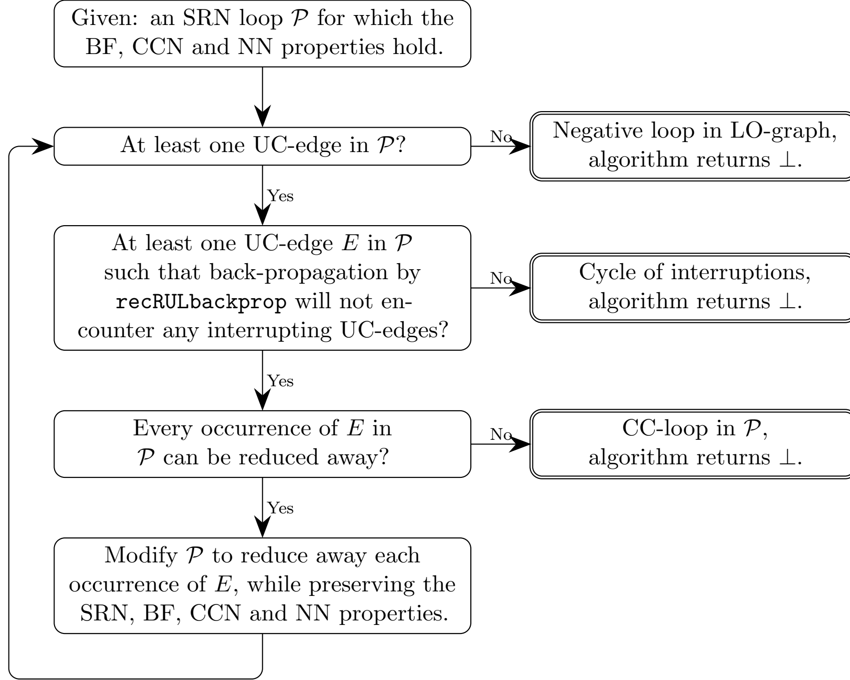


Figure 30: The general structure of the inductive argument in the proof of correctness for the RUL2020 DC-checking algorithm

the algorithm's processing of each UC-edge in $\mathcal{P}$ encounters an interrupting UC-edge. That would imply a cycle of interruptions which would ensure that the algorithm returns $\perp$. On the other hand, if there is some UC-edge E for which the algorithm would not encounter any interrupting UC-edge, then there are two cases. First, if back-propagation is blocked by encountering a CC-loop, then, because $\mathcal{P}$ is semi-reducible, that would ensure that the corresponding forward propagation would discover an extension sub-path for the relevant LC-edge, upon which discovery the algorithm would return $\perp$. On the other hand, if each occurrence of the UC-edge E in $\mathcal{P}$ can be reduced away, then modify $\mathcal{P}$ by replacing each sub-path used to reduce away an occurrence of E by an ordinary edge, thereby removing all occurrences of E in $\mathcal{P}$, while preserving the BF, CCN and NN properties. The result would be an SRN loop $\mathcal{P}$ for which the BF, CCN and NN properties hold, but having one fewer *distinct* UC-edges than before.

The following lemma is a generalization of a result by Morris (2006).

Lemma 1 ((Hunsberger, 2013)). *Let $\mathcal{P}$ be a semi-reducible path that contains a sub-loop, S, such that:*

1. $|S| \geq 0$*;*

2. *every extension sub-path in $\mathcal{P}$ that contains S, but whose corresponding*

lower-case edge is not in S, is breach-free (BF) (i.e., for each LC-edge $(A, c{:}x, C)$ in $\mathcal{P}$, its extension sub-path does not contain any occurrences of the corresponding upper-case edge $(C, C{:}{-y}, A)$); and

3. *no proper prefix of S is also the suffix of an extension sub-path in $\mathcal{P}$ whose corresponding LC-edge is not in S.*

Then the path, $\mathcal{P}'$, formed by extracting S from $\mathcal{P}$ is semi-reducible with $|\mathcal{P}'| \leq |\mathcal{P}|$.

Proof. The first property ensures that $|\mathcal{P}'| \leq |\mathcal{P}|$. For semi-reducibility, suppose that e is an occurrence of an LC-edge in $\mathcal{P}'$. Let $\mathcal{P}_e$ be the extension sub-path for e in $\mathcal{P}$. If $\mathcal{P}_e$ does not intersect S non-trivially, then $\mathcal{P}_e$ is also the extension sub-path for e in $\mathcal{P}'$. Given the third property, the only other way $\mathcal{P}_e$ could intersect with S is if it contained S. But then the path $\mathcal{P}'_e$ obtained by extracting S from $\mathcal{P}_e$ would satisfy $|\mathcal{P}'_e| \leq |\mathcal{P}_e| < 0$ and hence would contain the extension sub-path for e in $\mathcal{P}'$. By the second property, $\mathcal{P}_e$ (and hence $\mathcal{P}'_e$) is breach-free. Therefore, e could be reduced away by its extension sub-path $\mathcal{P}'_e$ in $\mathcal{P}'$. □

Lemma 2, below, states three properties of paths in STNU graphs that play important roles in the proof of correctness for the RUL2020 algorithm. It shows that any non-DC STNU graph must have an SRN loop $\mathcal{P}$ for which these three properties hold.

Lemma 2. *If $\mathcal{G}$ is the graph for an STNU that is not DC, then there exists a semi-reducible negative loop $\mathcal{P}$ in $\mathcal{G}$ such that all of the following properties hold.*

BF (Breach-Free): $\mathcal{P}$ is breach-free.

CCN (CC Negative*): For any consecutive occurrences of a contingent time-point C in $\mathcal{P}$, the sub-path $\mathcal{P}'$ from the first occurrence of C to the second has at least one prefix (including possibly the entire sub-path $\mathcal{P}'$) that has negative length.*

*NN (*No Naked *contingent time-points): Each occurrence of a contingent time-point C in $\mathcal{P}$ is either immediately preceded by the corresponding lower-case edge $(A, c{:}x, C)$ or immediately followed by the corresponding upper-case edge $(C, C{:}{-y}, A)$ (or both).*

Proof. Let $\mathcal{G}$ be the graph for an STNU that is not DC. Morris (2006) proved that $\mathcal{G}$ must contain an SRN loop $\mathcal{P}$. In addition, he proved that any breaches in $\mathcal{P}$ can be removed, as follows. Let $e = (A, c{:}x, C)$ be any LC-edge whose extension sub-path $\mathcal{P}_e$ in $\mathcal{P}$ contains a breach. If there are multiple such LC-edges, choose one whose extension sub-path is *not*

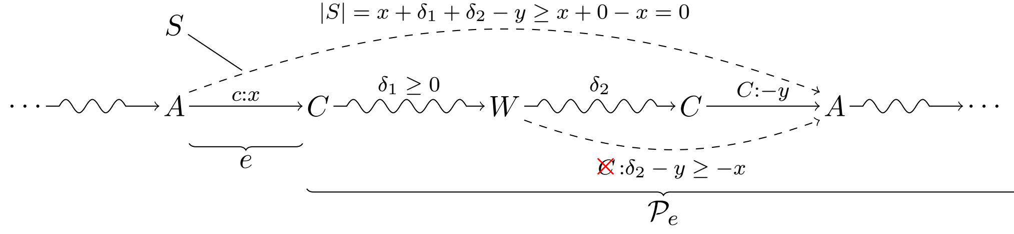
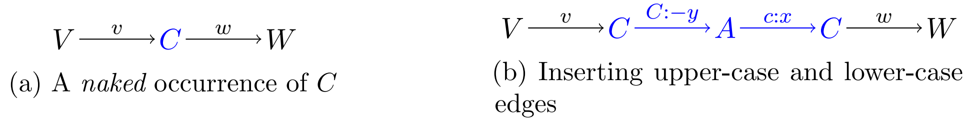


Figure 31: Removing a breach from an SRN loop by extracting the sub-loop S from A to A

nested within any other breach-containing extension sub-path. As illustrated in Figure 31, a breach $(C, C{:}{-y}, A)$ within $\mathcal{P}_e$ implies that there is a sub-loop S from the occurrence of A in e to the occurrence of A in the breach edge. Since $\mathcal{P}$ is semi-reducible, then, as shown in the figure, the edge from some time-point W to A generated by applying the path-transformation rules from Table 3 must have its upper-case label removed by the Label Removal rule, which implies that its length must satisfy $\delta_2 - y \geq -x$. In addition, $\delta_1 \geq 0$, since any proper prefix of an extension sub-path must be non-negative. But then $|S| = x + \delta_1 + \delta_2 - y \geq x + 0 - x = 0$. Thus, S satisfies the conditions of Lemma 1; hence extracting S from $\mathcal{P}$ preserves its semi-reducibility. Furthermore, since $|S| \geq 0$ and $\mathcal{P}$ is breach-free, extracting S from $\mathcal{P}$ might cause an extension sub-path to terminate earlier than it does in $\mathcal{P}$, but it cannot introduce a breach. Processing each breach in $\mathcal{P}$ in this way removes all breaches from $\mathcal{P}$ while preserving its semi-reducibility.

For the CCN property, suppose that $\mathcal{P}$ is a BF SRN loop, and S is any sub-path from one occurrence of a contingent time-point C in $\mathcal{P}$ to the next occurrence of C in $\mathcal{P}$. If every prefix of S (including S itself) has non-negative length, then S cannot contain a suffix of any extension sub-path because any such suffix must have negative length while also being a prefix of S, which is a contradiction. And since $\mathcal{P}$ is breach-free, S satisfies the conditions of Lemma 1. Hence, extracting S from $\mathcal{P}$ preserves its semi-reducibility and, as noted above, cannot introduce any breaches. Continuing in this way removes all sub-paths violating the CCN property, while preserving both the BF and semi-reducibility properties.



(a) A *naked* occurrence of C

(b) Inserting upper-case and lower-case edges

Figure 32: Replacing a *naked* occurrence of C by a two-edge path containing the upper-case and lower-case edges associated with C (cf. Proof of Lemma 2)

For the NN property, suppose that $\mathcal{P}$ is a BF SRN loop that also satisfies

the CCN property. Suppose further that, as illustrated in Figure 32a, C is an occurrence of a contingent time-point in $\mathcal{P}$ that is naked (i.e., is not preceded in $\mathcal{P}$ by the LC-edge $(A, c{:}x, C)$ and is not followed by the UC-edge $(C, C{:}{-y}, A)$). In the figure, note that the edges (V, v, C) and (C, w, W) may be any kind of edge (ordinary, lower-case or upper-case), just not the respective UC- or LC-edge associated with the contingent link (A, x, y, C). Let $\mathcal{P}^*$ be the path obtained by inserting the two-edge sub-path $(C, C{:}{-y}, A, c{:}x, C)$ into $\mathcal{P}$ as illustrated in Figure 32b. Notice that instead of extracting a non-negative sub-loop S as in Lemma 1, here we are inserting a negative sub-loop (from C to C) into $\mathcal{P}$, but the analysis is similar to show that it preserves the semi-reducible, BF and CCN properties.

For semi-reducibility and the BF property, suppose $e = (A', c'{:}x', C')$ is an LC-edge that precedes the occurrence of C in Figure 32a but whose extension sub-path $\mathcal{P}_e$ contains that occurrence of C as an interior point. Note that C' must not be the same as C because then the prefix of $\mathcal{P}_e$ from C' to C would violate the CCN property for $\mathcal{P}$, since every prefix of an extension sub-path must have non-negative length. But then the newly introduced upper-case edge $(C, C{:}{-y}, A)$ cannot be a breach edge for e. In addition, since $\mathcal{P}$ has no breaches, inserting the negative two-edge path might make the extension sub-path for e terminate earlier, but it cannot introduce a breach edge into $\mathcal{P}_e$. Similarly, the newly inserted lower-case edge $(A, c{:}x, C)$ cannot have any breaches in its extension sub-path in $\mathcal{P}^*$ because the presence of such a breach would imply the existence of a prefix in its extension sub-path from C to C which would violate the CCN property for $\mathcal{P}$. Therefore, the new path $\mathcal{P}^*$ is both BF and semi-reducible.

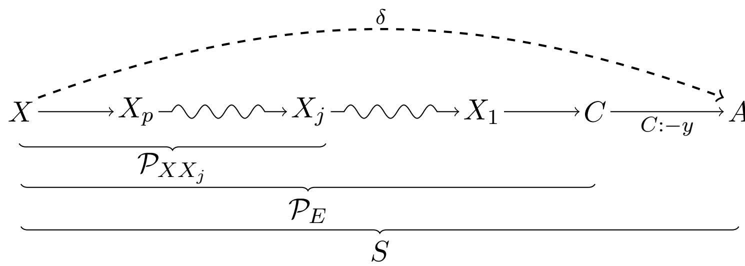
In addition, $\mathcal{P}^*$ also has the CCN property, as follows. First, any occurrence of C either preceding or following the one shown in Figure 32a could not cause a violation of the CCN property in $\mathcal{P}^*$ due to the CCN property holding for $\mathcal{P}$. Second, suppose consecutive occurrences of some other contingent time-point C' straddled the occurrence of C in Figure 32a. By the CCN property for $\mathcal{P}$, there must be a negative prefix of the path joining those two occurrences of C'. And the presence of the newly inserted negative two-edge path can only make that prefix more negative (if it intersects) or unchanged (if it doesn't intersect). □

Theorem 1. *Let $\mathcal{S}$ be any non-DC STNU. Then the new RUL2020 algorithm will return $\perp$ (i.e., will declare that $\mathcal{S}$ is not DC).*

Proof. Let $\mathcal{S}$ be any non-DC STNU. By Lemma 2, there exists an SRN loop $\mathcal{P}$ for which the BF, CCN and NN properties all hold. The rest of the proof is by induction, as illustrated previously in Figure 30. First, if $\mathcal{P}$ has no UC-edges, then it is a negative loop in the LO-graph. The algorithm would detect this negative loop when it next tried to compute or update a potential function for the LO-graph using Bellman-Ford or the `updatePotential` function.

Next, consider the back-propagation performed by the RUL2020 algorithm as it processes the UC-edges occurring in $\mathcal{P}$. Note that this back-propagation is not restricted to edges in $\mathcal{P}$, but explores edges from the entire LO-graph. If the processing of each UC-edge would encounter an interrupting UC-edge, whether in $\mathcal{P}$ or not, then the algorithm would detect a cycle of interruptions and return $\perp$. Note that the order in which the algorithm initially attempted to process the different UC-edges is irrelevant since processing one UC-edge only continues after all interrupting processes complete.

The rest of the proof focuses on the case where there is some UC-edge $E = (C, C{:}{-y}, A)$ in $\mathcal{P}$ for which the back-propagation from C along LO-edges (in the entire STNU graph) does not encounter any other interrupting UC-edge. Consider one such UC-edge E. Let $d\colon \mathcal{T} \mapsto \mathbb{R}$ be the distance function computed by the RUL2020 algorithm during its back-propagation from C along shortest paths in the LO-graph. (d is called `distFrom` in the pseudocode.) Again, it is important to stress that this back-propagation takes place in the entire LO-graph, not just along edges in $\mathcal{P}$. Furthermore, the sequence of edges immediately preceding any given occurrence of C in $\mathcal{P}$ may not be the same sequence of edges that the algorithm follows during its back-propagation from C in the LO-graph. For example, it may be that some sub-paths in $\mathcal{P}$ are not shortest paths in the LO-graph. Nonetheless, for each $X \in \mathcal{T}$ that is encountered during back-propagation from C, let $d(X)$ equal the length of the shortest encountered path from X to C; for all other X, let $d(X) = \infty$. Next, consider any particular occurrence of E in $\mathcal{P}$. (There may be many occurrences of E in $\mathcal{P}$.) And consider the sub-path $\mathcal{P}_E$ immediately preceding that occurrence of E in $\mathcal{P}$, defined as follows. Start at the time-point C, then follow the LO-edges backward from C in $\mathcal{P}$, until the first time-point X is encountered such that one of the following cases holds.



where $d(X) \geq \Delta_C$, $d(X_j) < \Delta_C$ for each $1 \leq j \leq p$, and $\delta = d(X) - y \geq \Delta_C - y = -x$

Figure 33: The scenario described in Case 1 in the proof of Theorem 1

Case 1: $d(X) \geq \Delta_C$. This case is illustrated in Figure 33. Since $d(X)$ is the length of the shortest path encountered from X to C during back-propagation along LO-edges from C in $\mathcal{S}$, and the path $\mathcal{P}_E$ is a path

from X to C in the LO-graph, it follows that every time point in $\mathcal{P}_E$ is explored by the algorithm's back-propagation from C in the LO-graph, even if not encountered in the same order as their appearance in $\mathcal{P}_E$, and even if the sub-paths of $\mathcal{P}_E$ are not necessarily shortest paths. Finally, $|\mathcal{P}_E| \geq d(X) \geq \Delta_C$.

Next, let X_j be any interior point (if any) along the path from X to C in $\mathcal{P}_E$, and let $\mathcal{P}_{XX_j}$ be the corresponding sub-path from X to X_j, also illustrated in Figure 33. Now, by construction, $d(X_j) < \Delta_C$. In addition, although the various sub-paths of $\mathcal{P}_E$ may not be shortest paths, the length of the sub-path from X to X_j—*in* $\mathcal{P}$—must satisfy $|\mathcal{P}_{XX_j}| \geq |\mathcal{P}_E| - d(X_j) \geq d(X) - d(X_j) > \Delta_C - \Delta_C = 0$. As a result, every prefix of $\mathcal{P}_E$, including $\mathcal{P}_E$ itself, has non-negative length.

Next, let S be the sub-path from X to A, shown in Figure 33, that consists of the path $\mathcal{P}_E$ followed by the UC-edge E. Then let $\mathcal{P}^*$ be the path obtained from $\mathcal{P}$ by replacing the sub-path S by the single edge (X, δ, A), shown as a dashed edge in Figure 33, where $\delta = d(X) - y \geq \Delta_C - y = -x$.

Note that $|\mathcal{P}^*| \leq |\mathcal{P}|$, since $\delta = d(X) - y \leq |\mathcal{P}_E| - y = |S|$. To show that $\mathcal{P}^*$ is semi-reducible, we argue as in the proof of Lemma 1. Let e be any LC-edge that precedes $\mathcal{P}_E$ in $\mathcal{P}$, and let $\mathcal{P}_e$ be the extension sub-path for e in $\mathcal{P}$. If $\mathcal{P}_e$ does not intersect S non-trivially, then $\mathcal{P}_e$ is also the extension sub-path for e in $\mathcal{P}^*$. On the other hand, since all suffixes of extension sub-paths necessarily have negative length, the fact that all proper prefixes of S are non-negative ensures that $\mathcal{P}_e$ cannot have a suffix that is a proper prefix of S. Therefore, the only other way $\mathcal{P}_e$ can intersect non-trivially with S is if it contains S. Let $\mathcal{P}_e^*$ be the path obtained by replacing S by the edge (X, δ, A). Then $\delta \leq |S|$ ensures that that $|\mathcal{P}_e^*| \leq |\mathcal{P}_e| < 0$. And since $\mathcal{P}$ is breach-free, so too is $\mathcal{P}_e^*$. Thus, e can be reduced away by $\mathcal{P}_e^*$ in $\mathcal{P}^*$. Hence, $\mathcal{P}^*$ is semi-reducible and also breach-free.

As for the CCN property, let C_a and C_b be two successive occurrences of the same contingent time-point in $\mathcal{P}^*$, and let $\mathcal{P}_{CC}$ be the sub-path of $\mathcal{P}^*$ from C_a to C_b. If the newly inserted edge (X, δ, A) is not part of $\mathcal{P}_{CC}$, then $\mathcal{P}_{CC}$ is in $\mathcal{P}$ and it has a negative prefix courtesy of the CCN property for $\mathcal{P}$. Otherwise, there are two cases to consider.

Case A: There is no occurrence of that contingent time-point in S. In this case, the path from C_a to C_b in $\mathcal{P}$ must have a negative prefix $\mathcal{P}_{ab}$ in $\mathcal{P}$. If $\mathcal{P}_{ab}$ terminates at or before X in $\mathcal{P}$, then $\mathcal{P}_{ab}$ is also a negative prefix for $\mathcal{P}_{CC}$ in $\mathcal{P}^*$. Otherwise, if $\mathcal{P}_{ab}$ terminates somewhere in the interior of S, then the prefix of $\mathcal{P}_{ab}$ that terminates at X must also be negative, since the portion of $\mathcal{P}_{ab}$ after X is a non-negative proper prefix of S. Otherwise, $\mathcal{P}_{ab}$ must contain

all of S, in which case replacing S by the edge (X, δ, A) in $\mathcal{P}_{ab}$ produces a negative prefix of $\mathcal{P}_{CC}$ in $\mathcal{P}_{CC}$, since $\delta \leq |S|$.

Case B: There is an occurrence C_c of the same contingent time-point within S.

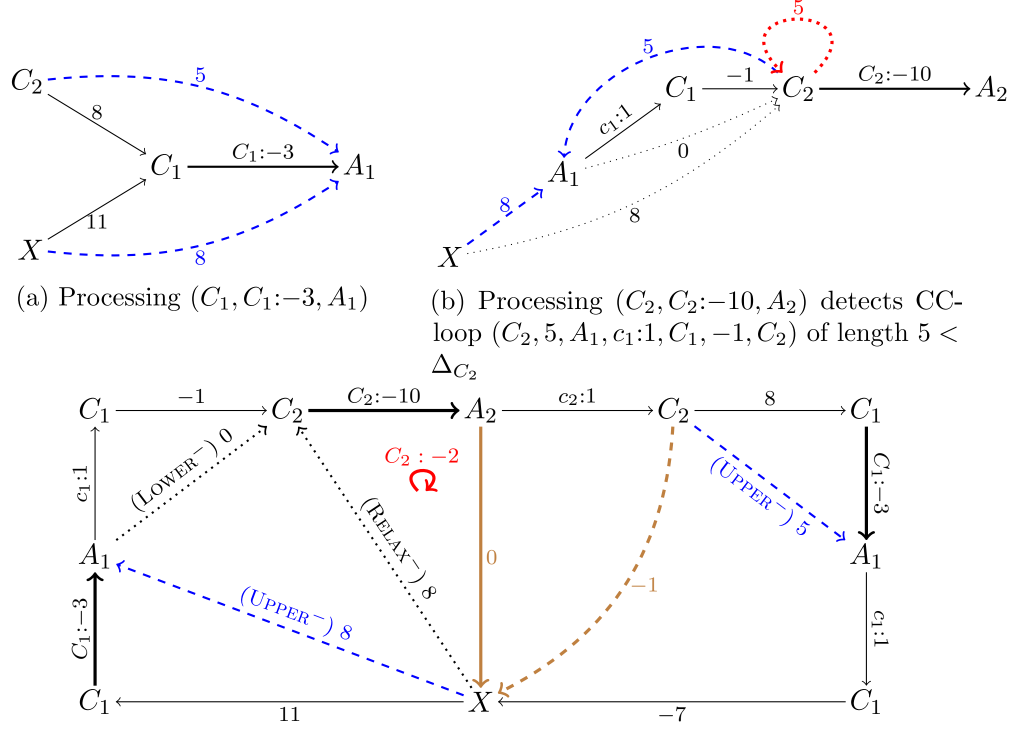
In this case, any negative prefix of the path from C_a to C_c in $\mathcal{P}$ can be cropped at X (if necessary) to yield a negative prefix of the path from C_a to C_b in $\mathcal{P}^*$, since every proper prefix of S is non-negative, and $\delta \leq |S|$.

Therefore, the CCN property must hold for $\mathcal{P}^*$.

Finally, for the NN property, suppose that replacing $\mathcal{P}_E$ by the edge (X, δ, A) violated the NN property for $\mathcal{P}^*$. That could only happen at the end-points, X or A. However, for X to be naked in $\mathcal{P}^*$ would require X to be a contingent time-point and the edge terminating at X *not* being the corresponding LC-edge. But the edge emanating from X in S must be an LO-edge and hence cannot be the corresponding UC-edge, which would imply that X was naked in $\mathcal{P}$, a contradiction. Similarly, for A to be naked in $\mathcal{P}^*$ would require A to be contingent and the edge emanating from A *not* being the corresponding UC-edge. But the edge preceding A in S is a UC-edge, which would imply that A was naked in $\mathcal{P}$, a contradiction. Thus, the NN property holds for $\mathcal{P}^*$.

Given the above analysis, replacing the path S by the single edge (X, δ, A) results in a semi-reducible negative loop $\mathcal{P}^*$ for which the BF, CCN and NN properties all hold, but from which the occurrence of the UC-edge E has been removed.

Case 2: $X \equiv C$ and $d(C) < \Delta_C$. For example, recall the SRN loop from Figure 18, repeated here for convenience as Figure 34. First, processing the UC-edge $(C_1, C_1{:}{-3}, A_1)$ generated the edges $(C_2, 5, A_1)$ and $(X, 8, A_1)$, shown as blue, dashed edges in Figure 34a. Next, back-propagation from C_2 in Figure 34b yielded the following distances: $d(C_1) = -1$, $d(A_1) = 0$, $d(X) = 8$, $d(C_2) = 5$. Finally, following the edges in $\mathcal{P}$ backward from C_2—after having reduced away all occurrences of the UC-edge $(C_1, C_1{:}{-3}, A_1)$—yields the path shown in Figure 35. Notice that some time-points (e.g., C_1 and A_1) appear more than once in this sequence, and not all sub-paths are shortest paths to C_2 (e.g., the sub-path from the first occurrence of A_1 to C_2 has length 2, while the sub-path from the second occurrence of A_1 to C_2 has length 0); however, the shortest-path distance from each time-point in this sequence to C_2 is less than $\Delta_{C_2} = 9$. Then, forward propagation from C_2 discovers the extension sub-path $(C_2, 5, A_1, c_1{:}1, C_1, -7, X)$ which can be used to reduce away the LC-edge $(A_2, c_2{:}1, C_2)$, ensuring that there is a



(c) Forward propagation from C_2 to A_1 to C_1 to X detects an extension sub-path (dashed, brown) for the LC-edge $(A_2, c_2{:}1, C_2)$ enabling it to be reduced away (solid, brown), yielding a negative loop in the OU-graph (red), implying that the STNU is not DC

Figure 34: The RUL2020 algorithm's processing of the STNU from Figure 18

negative loop from A_2 to A_2 in the OU-graph. Thus, the algorithm declares the network to be non-DC.

More generally, this case is illustrated in Figure 36. By the CCN and BF properties for $\mathcal{P}$, the path $\mathcal{P}_E$ from C to C must have a negative prefix. If that negative prefix is $\mathcal{P}_E$ itself, then $0 > |\mathcal{P}_E| \geq d(C)$ which causes the algorithm to immediately return $\perp$ (see Line 12 of `OneStepBackProp`, Algorithm 12). Otherwise, the negative proper prefix of $\mathcal{P}_E$ implies that the LC-edge $(A, c{:}x, C)$ must have a breach-free extension sub-path in $\mathcal{P}_E$ that can be used to reduce it away, represented by the dashed edge from A to X_j of length $\delta < x$. Because $d(X_j) < \Delta_C$, there must be some path in the LO-graph from X_j to C of length $d(X_j) < \Delta_C = y - x$ which could be used to generate an edge from X_j to C of length $d(X_j) < \Delta_C$. These two edges, together with the UC-edge $(C, C{:}{-y}, A)$, compose a negative loop from A to A in the OU-graph, which implies that the network is non-DC. Of course, the algorithm doesn't do all of this edge generation. Instead, it

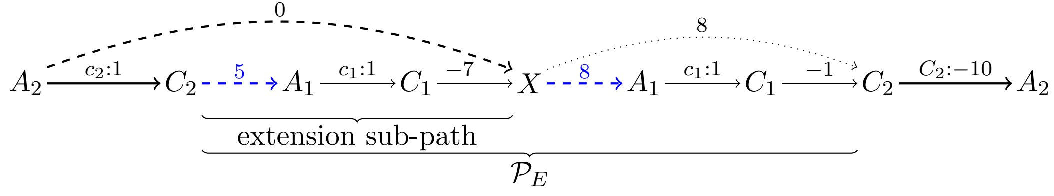
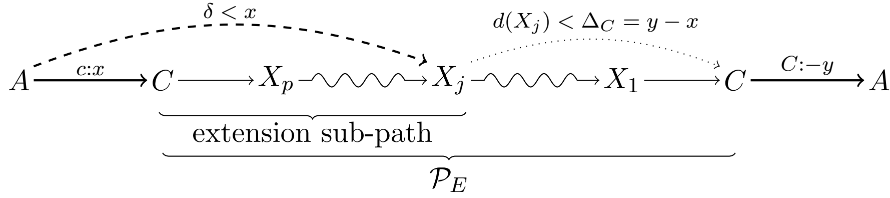


Figure 35: Sample scenario for Case 2 in the proof of Theorem 1



where $d(C) < \Delta_C$, $d(X_j) < \Delta_C$, and $\delta + d(X_j) - y < x + (y - x) - y = 0$

Figure 36: The general scenario for Case 2 in the proof of Theorem 1

simply detects the extension sub-path during its forward propagation from C. Because it only visits time-points Y for which $d(Y) < \Delta_C$, if it finds such an extension sub-path, it necessarily follows that the network must not be DC. Hence, the algorithm immediately returns $\perp$ (see Lines 24-26 of `recRULbackprop`, Algorithm 11).

Note. Given the CCN property for $\mathcal{P}$, backtracking from C along LO-edges in $\mathcal{P}$ cannot encounter a naked contingent time-point. This is important because the restriction $Q \in \mathcal{T}_X$ in the Relax$^-$ rule would cause the algorithm's back-propagation from C to be blocked by a naked contingent time-point Q, even if $d(Q) < \Delta_C$, thereby threatening to make $\mathcal{P}_E$ not well defined. In view of this, Cases 1 and 2 above are the only possible cases and $\mathcal{P}_E$ is well defined.

Overall, then, if Case 2 is ever encountered, the algorithm will immediately return $\perp$. Otherwise, Case 1 can be used to reduce away—and remove from $\mathcal{P}$—each occurrence of the UC-edge E while preserving the SRN, BF, CCN and NN properties. After all occurrences of E have been removed from $\mathcal{P}$, the number of *distinct* UC-edges in $\mathcal{P}$ has been reduced by one, concluding the inductive case. □

References

Bannister, M. J., & Eppstein, D. (2012). Randomized Speedup of the Bellman–Ford Algorithm. In *9th Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pp. 41–47, doi: 10.1137/1.9781611973020.6.

Cairo, M., Hunsberger, L., & Rizzi, R. (2018). Faster dynamic controllablity checking for simple temporal networks with uncertainty. In *25th Int. Symp. on Temporal Representation and Reasoning (TIME-2018)*, Vol. 120 of *LIPIcs*, pp. 8:1–8:16, doi: 10.4230/LIPIcs.TIME.2018.8.

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

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

Hunsberger, L. (2013). Magic Loops in Simple Temporal Networks with Uncertainty–Exploiting Structure to Speed Up Dynamic Controllability Checking. In *5th Int. Conf. on Agents and Artificial Intelligence (ICAART-2013)*, Vol. 2, pp. 157–170.

Hunsberger, L. (2014a). A faster algorithm for checking the dynamic controllability of simple temporal networks with uncertainty. In *6th Int. Conf. on Agents and Artificial Intelligence (ICAART-2014)*.

Hunsberger, L. (2014b). Magic Loops and the Dynamic Controllability of Simple Temporal Networks with Uncertainty. In Filipe, J., & Fred, A. (Eds.), *Agents and Artificial Intelligence*, Vol. 449 of *Communications in Computer and Information Science (CCIS)*, pp. 332–350, doi: 10.1007/978-3-662-44440-5_20.

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

Hunsberger, L. (2015b). New techniques for checking dynamic controllability of simple temporal networks with uncertainty. In Duval, B., van den Herik, J., Loiseau, S., & Filipe, J. (Eds.), *6th Int. Conf. Agents and Artificial Intelligence (ICAART-2014), Revised Selected Papers*, Vol. 8946 of *Lecture Notes in Computer Science*, pp. 170–193.

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

Morris, P. (2014). Dynamic controllability and dispatchability relationships. In *Integration of AI and OR Techniques in Constraint Programming. CPAIOR 2014.*, Vol. 8451 of *LNCS*, pp. 464–479. Springer, doi: 10.1007/978-3-319-07046-9_33.

Morris, P. H., & Muscettola, N. (2005). Temporal dynamic controllability revisited. In *20th National Conf. on Artificial Intelligence (AAAI-2005)*, pp. 1193–1198.

Morris, P. H., Muscettola, N., & Vidal, T. (2001). Dynamic control of plans with temporal uncertainty. In *17th Int. Joint Conf. on Artificial Intelligence (IJCAI-2001)*, pp. 494–502.

Nilsson, M., Kvarnstrom, J., & Doherty, P. (2014). Incremental dynamic controllability in cubic worstcase time. In *21st Int. Symp. on Temporal Representation and Reasoning (TIME-2014)*.

Object Management Group (OMG) (2007). Business process definition metamodel (bpdm), Beta 1. http://www.omg.org.

Posenato, R. (2020). The CSTNU toolset. version 3.0. http://profs.scienze.univr.it/∼posenato/software/cstnu.

Ramalingam, G., Song, J., Joskowicz, L., & Miller, R. E. (1999). Solving Systems of Difference Constraints Incrementally. *Algorithmica*, *23*(3), 261–275, doi: 10.1007/PL00009261.

Algorithm 4: More detailed pseudocode for Morris' 2006 algorithm

Input: $\mathcal{G}$ // An STNU graph

Output: $\top$, if $\mathcal{G}$ is DC; $\bot$, otherwise // Side Effect: Modifies $\mathcal{G}$

1 **for** $(i := 1\ to\ k)$ **do** // Outer Loop: k iterations, one for each level of nesting of extension sub-paths

2 h := $BellmanFord(\mathcal{G}_{ou})$

3 **if** $(h == \bot)$ **then return** $\bot$

4 newOrdEdges := {}, newUCEdges := {}

5 **foreach** $((A, c{:}x, C)\ in\ \mathcal{G})$ **do** // Inner Loop: search shortest *allowable* paths emanating from C

6 $\mathcal{Q}$:= a new priority queue

7 $\mathcal{Q}.insert(C, 0)$

8 **while** $(\neg\mathcal{Q}.empty())$ **do**

9 (U, nonNegCU) := $\mathcal{Q}.extractMinNode()$ // nonNegCU = adjusted distance from C to U

10 realCU := nonNegCU - h[C] + h[U] // realCU = actual distance from C to U in $\mathcal{G}$

11 **if** (realCU < 0) **then**

// Case 1: Found full extension sub-path for generating ordinary edge from A to U

12 newAU := x + realCU // Application of Lower-Case Rule

13 **if** (newAU < $\mathcal{G}.getOrdEdgeWt(A, U)$) **then**

newOrdEdges.add(A,newAU,U)

14 **else** // Case 2: Not at end of ext. sub-path; must propagate forward from U along OU-edges.

15 **foreach** $((U, \delta_{UV}, V) \in \mathcal{E}_o)$ **do** // Propagate forward from U along ordinary edges

16 nonNegCUV := nonNegCU + (δ_{UV} + h[U] - h[V])

17 $\mathcal{Q}.insertOrDecreaseKeyIfSmaller(V, \texttt{nonNegCUV})$

18 **foreach** $((U, C'{:}w, A') \in \mathcal{E}_u)$ **do** // Propagate forward from U along UC-edges

19 **if** $(C \equiv C')$ **then** continue // Can't combine LC, UC edges from same cont. link

20 realCUA′ := realCU + w

21 newAA′ := x + realCUA′ // real weight on possible new edge from A to A′

22 nonNegCUA′ := nonNegCU + (w + h[U] - h[A′])

23 x' := lower bound on duration of cont. link associated with C'

24 **if** ((realCUA′ < 0) and (newAA′ < $-x'$)) **then** // Will gen. new UC-edge

25 currAA′ := $\mathcal{G}.getUCEdgeWt(A, A', C')$ // ∞ if no UC edge AA′

26 **if** (newAA′ < currAA′) **then**

newUCEdges.add(A, C':newAA′$, A'$)

27 **else** // Either not at end of extension sub-path or will generate ordinary edge

28 $\mathcal{Q}.insertOrDecreaseKeyIfSmaller(A', \texttt{nonNegCUA}')$

29 **if** ((*newOrdEdges.empty()* and *newUCEdges.empty()*)) **then return** $\top$

30 **foreach** $((A, C'{:}w, A') \in newUCEdges)$ **do** $\mathcal{G}.insertOrUpdateUCEdge(A, C'{:}w, A')$

31 **foreach** $((A, \delta, X) \in newOrdEdges)$ **do** $\mathcal{G}.insertOrUpdateOrdEdge(A, \delta, X)$

32 **if** $(BellmanFord(\mathcal{G}_{ou}) \neq \bot)$ **then return** $\top$ // If $\mathcal{G}_{ou}$ still consistent, $\mathcal{G}$ must be DC

33 **else return** $\bot$

Algorithm 5: Morris' 2014 DC-checking algorithm for STNUs

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph in normal form

Output: $\top$, if $\mathcal{G}$ is DC; $\perp$, otherwise // Side Effect: Modifies $\mathcal{G}$

1 **foreach** (*negative node* X *in* $\mathcal{T}$) **do**

2   **if** (`DCbackprop`$(\mathcal{G}, X) == \perp$) **then return** $\perp$

3 **return** $\top$

Algorithm 6: The `DCbackprop` function used in Morris' 2014 DC-checking algorithm

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph in normal form; X, a negative node in $\mathcal{T}$

Output: $\top$, if all negative edges incoming to X can be reduced away, $\bot$ otherwise

// Side Effect: Modifies $\mathcal{G}$

1. **if** *(Ancestor call with same X)* **then return** $\bot$ // if state is started...
2. **if** *(Prior call with this X completed)* **then return** $\top$ // if state is finished...
3. `dist`$[X] := 0$
4. **foreach** $(Y \in \mathcal{T} \mid Y \not\equiv X)$ **do** `dist`$[Y] := \infty$ // Initialize distance function d
5. $\mathcal{Q}$:= a new priority queue
6. **if** *(X is ATP for some $(Y, Y{:}\delta, X) \in \mathcal{E}_u$)* **then** // Case 1: X has one incoming neg. UC-edge
7. $\mathcal{Q}.insert(Y, \delta)$
8.   `dist`$[Y] := \delta$
9. **else** // Case 2: X has one or more incoming ordinary neg. edges
10.   **foreach** $((Y, \delta, X) \in \mathcal{E}_o \mid \delta < 0)$ **do**
11. $\mathcal{Q}.insert(Y, \delta)$
12.   `dist`$[Y] := \delta$
13. **while** $(\neg\mathcal{Q}.empty())$ **do**
14. $U := \mathcal{Q}.extractMinNode()$
15.   **if** (`dist`$[U] \geq 0$) **then** $\mathcal{G}.insertOrUpdateOrdEdge(U,$ `dist`$[U], X)$
16.   **else**
17.   **if** *((U is a neg. node)* `and` *(*`DCbackprop`$(\mathcal{G}, U) = \bot$*))* **then return** $\bot$
18.   **foreach** $((V, \alpha, U) \in \mathcal{E}_o \cup \mathcal{E}_\ell \mid \alpha \geq 0)$ **do**
                // If X is activation tp for cont link, then cannot back-prop along LC-edge for that cont link
19.   **if** *((($V, \alpha, U) \in \mathcal{E}_\ell$) **and** ($V \equiv X$))* **then** continue
20.   `newKey` := `dist`$[U] + \alpha$
21. $\mathcal{Q}.insertOrDecreaseKeyIfSmaller(V,$ `newKey`$)$
22.   `dist`$[V] := \mathcal{Q}.key(V)$
23. **return** $\top$

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

Table 4: The edge-generation rules for the RUL$^-$ algorithm

Algorithm 7: RUL-DC-check, the RUL$^-$ DC-checking algorithm

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph

Output: $\top$, if $\mathcal{G}$ is DC; $\bot$, otherwise // Side Effect: Modifies $\mathcal{G}$

1 $h :=$ BellmanFord ($\mathcal{G}_{\ell o}$) // Bellman-Ford on edges in $\mathcal{E}_o \cup \mathcal{E}_\ell$

2 **if** *($h == \bot$)* **then return** $\bot$

3 $\mathcal{U} := \mathcal{T}_C$ // $\mathcal{U}$ contains the contingent tps whose processing is unfinished, whether started or not

4 $S :=$ a new stack // S is the stack of contingent tps that have started to process

5 S.push(any element of $\mathcal{U}$) // Push arbitrary element of $\mathcal{U}$ onto S, while keeping it in $\mathcal{U}$

6 **while** *(S is not empty)* **do**

7     $(A, x, y, C) :=$ contingent link for contingent tp from top of S // But don't pop it from S

8     $\mathcal{G} :=$ CloseRelaxLower ($\mathcal{G}, h, C, y - x$) // Generate new edges terminating at C; insert into $\mathcal{G}$

9     $\mathcal{G} :=$ ApplyUpper ($\mathcal{G}, A, x, y, C$) // Generate new edges terminating at A; insert into $\mathcal{G}$

10     $h :=$ UpdatePotential ($\mathcal{G}_{\ell o}, h, A$) // Update potential function in response to new edges

11   **if** *($h == \bot$)* **then return** $\bot$

    // Check for possible interruption due to an encounter with another upper-case edge $(C', C'{:}y', A')$

12   **if** *($\exists C' \in \mathcal{U}$ and an edge $(A', w, C) \in \mathcal{E}_o$ such that $w < \Delta_C = y - x$)* **then**

13   **if** *($C' \in S$)* **then return** $\bot$ // Negative cycle of interruptions

14   S.push(C') // Otherwise, prepare to process C' as an interruption of C

15   **else**

16         $\mathcal{U} := \mathcal{U} \setminus \{C\}$ // C has finished, so remove it from $\mathcal{U}$ and S

17   S.pop()

18   **if** *($\mathcal{U}$ is non-empty and S is empty)* **then** // More contingent tps to process

19   S.push($\mathcal{U}$.top())

20 **return** $\top$

Algorithm 8: Functions to initialize/update a lower-bound potential function

1 **function** `BellmanFord` ($\mathcal{G}$):

  Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph

  Output: h, a potential function such that $h(V) \geq h(W) - w$ for each edge $(V, w, W) \in \mathcal{G}$, unless $\mathcal{G}$ has negative cycle

2   **foreach** $V \in \mathcal{T}$ **do** $h(V) := 0$

3   **for** $i = 1$ ***to*** $|\mathcal{T}| - 1$ **do**

4   **foreach** *edge* $(V, w, W) \in \mathcal{G}$ **do**

5       $h(V) := \max\{h(V), h(W) - w\}$ // Ensures that $h(V) \geq h(W) - w$

  // Checks if there is a negative cycle

6   **foreach** *edge* $(V, w, W) \in \mathcal{G}$ **do**

7   **if** $h(V) < h(W) - w$ **then return** $\bot$

8   **return** h // Potential function

9 **function** `UpdatePotential` ($\mathcal{G}, h, A$):

  Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; h, a potential function such that $h(Y) - h(X) \leq \delta$ for all edges except possibly where $Y \equiv A$; $A \in \mathcal{T}$

  Output: h', a potential function such that $h'(V) \geq h'(W) - w$ for each $(V, w, W) \in \mathcal{E}_o \cup \mathcal{E}_\ell$ unless $\mathcal{G}_{\ell o}$ has a negative cycle

10   $h' := vectorCopy(h)$

11   $\mathcal{Q}$:= new priority queue

  // For each X, $\mathcal{Q}.key(X)$ = negative of the absolute amount $h(X)$ must change to restore solution

12   $\mathcal{Q}.insert(A, 0)$

13   **while** $\mathcal{Q}$ *is not empty* **do**

14     $W := \mathcal{Q}.extractMinNode()$

15   **foreach** $(V, w, W) \in \mathcal{E}_o \cup \mathcal{E}_\ell$ **do**

16   **if** $(h'(V) < h'(W) - w)$ **then**

17         $h'(V) := h'(W) - w$

         // newKey = negative of the amount potential function must change

18   `newKey` $:= h(V) - h'(V)$

19   **if** $(\mathcal{Q}.insertOrDecreaseKeyIfSmaller(V, \texttt{newKey}) == \bot)$ **then**

20   **return** $\bot$ // Negative cycle detected

21   **return** h'

Algorithm 9: Constraint-propagation functions for RUL-DC-check algorithm

1 **function** `ApplyRelaxLower` $(\mathcal{G}, W, C)$:
  Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; $W \in \mathcal{T}$; $C \in \mathcal{T}_C$
  Output: The set of all edges (V, v, C) obtained by applying RELAX$^-$ or LOWER$^-$ to edges $(V, \delta_{VW}, W) \in \mathcal{E}_o \cup \mathcal{E}_\ell$ together with $(W, \delta_{WC}, C) \in \mathcal{E}_o$. ($W$ fixed.)
2   $\delta_{WC} := \mathcal{G}.getOrdEdgeWt(W, C)$ // $\delta_{WC} = \infty$ if no pre-existing ordinary edge
3   **if** $(\delta_{WC} \geq \Delta_C)$ **then return** $\emptyset$ // RELAX$^-$ and LOWER$^-$ do not apply
4   **else if** $(W \in \mathcal{T}_C)$ **then return** $\{(A_W, x_W + \delta_{WC}, C)\}$ // Apply LOWER$^-$
5   **else return** $\{(V, \delta_{VW} + \delta_{WC}, C) \mid (V, \delta_{VW}, W) \in \mathcal{E}_o, V \not\equiv C\}$ // Apply RELAX$^-$

6 **function** `CloseRelaxLower` $(\mathcal{G}, h, C)$:
  Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; h, a potential function for $\mathcal{G}_{\ell o}$; $C \in \mathcal{T}_C$
  Pre: $h(W) - h(V) \leq \delta_{VW}$ for each $(V, \delta_{VW}, W) \in \mathcal{E}_o \cup \mathcal{E}_\ell$
  Post: All new ordinary edges (V, v, C) incoming to C that can be generated by (recursive) applications of RELAX$^-$ and LOWER$^-$ rules have been added to $\mathcal{E}_o$
7   $\mathcal{Q}$:= new priority queue
8   **foreach** $((W, \delta_{WC}, C) \in \mathcal{E}_o)$ **do**
9     $\mathcal{Q}.insert(W, h(W) + \delta_{WC})$
10 **while** $(\neg\mathcal{Q}.empty())$ **do**
11    $W := \mathcal{Q}.extractMinNode()$
12   **foreach** $((V, v, C) \in ApplyRelaxLower\ (\mathcal{G}, W, C))$ **do**
13   `currVC` $:= \mathcal{G}.getOrdEdgeWt(V, C)$ // `currVC` $= \infty$ if no existing ordinary edge
14   **if** $(v <$ `currVC`$)$ **then** $\mathcal{G}.insertOrUpdateOrdEdge(V, v, C)$ // Modifies $\mathcal{G}$
15   `newKey` $:= h(V) + \min\{v,$ `currVC`$\}$
16      $\mathcal{Q}.insertOrDecreaseKeyIfSmaller(V,$ `newKey`$)$
17 **return** $\mathcal{G}$

18 **function** `ApplyUpper` $(\mathcal{G}, C)$:
  Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; C, a cont. time-point for cont. link (A, x, y, C)
  Output: All new edges (V, w, A) obtained by applying UPPER$^-$ to any ordinary edge (V, v, C) together with the UC-edge $(C, C{:}{-y}, A)$ have been added to $\mathcal{E}_o$
19 **foreach** $((V, v, C) \in \mathcal{E}_o)$ **do**
20   `origWt` $:= \mathcal{G}.getOrdEdgeWt(V, A)$ // `origWt` $= \infty$ if no edge from V to A in $\mathcal{E}_o$
21   **if** $v < \Delta_C$ **then** `newWt` $:= \min\{$`origWt`$, -x\}$ // Non-length-preserving case
22   **else** `newWt` $:= \min\{$`origWt`$, v - y\}$ // Length-preserving case
23    $\mathcal{G}.insertOrUpdateOrdEdge(V,$ `newWt`$, A)$
24 **return** $\mathcal{G}$

Algorithm 10: `recRULdcCheck`, new recursive alg. that uses (most of) the RUL^- rules

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph with $n = |\mathcal{T}|$ and $k = |\mathcal{E}_u| = |\mathcal{E}_\ell|$

Output: $\top$, if $\mathcal{G}$ is DC; $\bot$, otherwise. // Side Effect: Modifies $\mathcal{G}$

```
1 globalInfo := new globalInfo struct      // Fields: potFunc, status (see below)
2 globalInfo.potFunc := BellmanFord (G_ℓo)  // Potential Func. is solution for
  LO-graph
3 if (globalInfo.potFunc == ⊥) then return ⊥
  // Initialize vector showing the processing status of each upper-case edge
4 globalInfo.status := [unstarted, unstarted, ..., unstarted]
  // k-vector
5 foreach ((C, C:−y, A) ∈ E_u) do
6   if (recRULbackprop (G, (C, C:−y, A), globalInfo) == ⊥) then
      return ⊥
7 return ⊤
```

Algorithm 11: The new `recRULbackprop` algorithm

Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, STNU graph; $\mathbf{E} = (C, C{:}{-y}, A) \in \mathcal{E}_u$; `globalInfo`

Output: $\top$, if $\mathbf{E}$ can be reduced away without cycle of recursive calls; $\perp$, otherwise.

// Side Effect: Modifies contents of $\mathcal{G}$ and the `globalInfo` data structure

1 `status` := `globalInfo.status` // Vector that gives processing status of each UC-edge

2 h := `globalInfo.potFunc` // Potential function, a solution to LO-graph

3 **if** (`status`($\mathbf{E}$) == `started`) **then return** $\perp$ // Cycle of recursive calls detected

4 **if** (`status`($\mathbf{E}$) == `finished`) **then return** $\top$ // This UC-edge already fully processed

5 `status`($\mathbf{E}$) := `started`

6 $\Delta_C := y - x$ // x = lower bound of contingent link corresponding to $\mathbf{E}$

7 `localInfo` := new *localInfo* struct // fields: `CC_loop?`, `unstarted-UCEs`, `dist-from`

8 `localInfo.CC_loop?` := $\perp$ // Will be set to $\top$ if loop found from C to C of length $< \Delta_C$

// `localInfo.unstarted-UCEs` := $\{(\mathbf{E}_X, X), \ldots\}$ will be initialized by `OneStepBackProp`

9 `localInfo.distFrom` := $[\infty, \ldots, \infty]$ // n vector that will give distance from each TP X to C

10 $\mathcal{Q}$:= a new priority queue

11 **foreach** ($(X, \delta_{XC}, C) \in \mathcal{E}_o$) **do** // Initialize queue

12     $\mathcal{Q}.insert(X, h(X) + \delta_{XC})$ // $key(X) = h(X) + \delta_{XC}$ = adjusted distance from X to C

13 `continue?` := $\top$

14 **while** (`continue?`) **do**

15   **if** (`OneStepBackProp`($\mathcal{G}, C, \mathcal{Q}$, `globalInfo`, `localInfo`) == $\perp$) **then return** $\perp$

16   **if** (`localInfo.unstarted-UCEs != {}`) **then**

        // Recursively process each unstarted UC-edge $\mathbf{E}_X$ that was encountered by `OneStepBackProp`

17   **foreach** ($(\mathbf{E}_X, X) \in$ `localInfo.unstarted-UCEs`) **do**

18   **if** (`rec-RUL-back-prop`($\mathcal{G}, \mathbf{E}_X$, `globalInfo`) == $\perp$) **then return** $\perp$

        // Re-initialize the queue for the next pass through the WHILE loop

19         $\mathcal{Q}.clear()$ // Clear the queue

20   **foreach** ($(\mathbf{E}_X, X) \in$ `localInfo.unstarted-UCEs`) **do**

21             $\mathcal{Q}.insert(X,$ `localInfo.distFrom`$[X]$ + `globalInfo.potFunc`$[X])$

22   `localInfo.distFrom`$[X] := \infty$ // So X handled properly by next call to `OneStepBackProp`

23   **else** `continue?` := $\perp$

24 **if** (`localInfo.CC_loop?` **and**

25   `fwdPropNotDC`($\mathcal{G}$, C, Δ_C, `localInfo.distFrom`, `globalInfo.potFunc`)) **then**

26   **return** $\perp$

// Apply Upper$^-$ rule to edges implicit in `distFrom` vector, and insert resulting edges into graph

27 `addedEdge?` := $\perp$

28 **foreach** ($X \in \mathcal{T}$ *such that* $X \not\equiv C$) **do**

29   δ_{XC} := `localInfo.distFrom`$[X]$

30   **if** ($\Delta_C \leq \delta_{XC} < \infty$) **then** // Length-preserving portion of Upper$^-$ rule

31         $\mathcal{G}.insertOrUpdateOrdEdge(X, \delta_{XC} - y, A)$ // Upper$^-$ to (X, δ_{XC}, C) and $(C, C{:}{-y}, A)$

32   `addedEdge?` := $\top$

33 **if** (`addedEdge?`) **then**

34   `globalInfo.potFunc` := `updatePotential`($\mathcal{G}, A$, `globalInfo.potFunc`)

35 **if** (`globalInfo.potFunc` == $\perp$) **then return** $\perp$

36 `status`$[\mathbf{E}]$:= `finished` **return** $\top$

Algorithm 12: `OneStepBackProp`

```
   Input: G = (T, E), STNU graph; E = (C, C:−y, A) ∈ E_u; Q, priority
          queue; globalInfo (see below); localInfo (see below)
   Output: ⊥, iff back-propagating from C reveals STNU to be non-DC
   // Side Effect: Modifies contents of localInfo data structure
1  h := globalInfo.potFunc            // Potential function, a solution to LO-graph
2  status := globalInfo.status        // Vector that gives processing status of each
                                      //   UC-edge
3  localInfo.unstarted-UCEs := {}     // Accumulates unstarted UC-edges seen
                                      //   during back-prop
4  while (not(Q.empty())) do
5     (X, key(X)) := Q.extractMinNode()   // key(X) = adjusted distance from X
                                          //   to C
6     δ_XC := key(X) − h(X)               // δ_XC = (actual) shortest distance from X to C
7     E_X := G.getUCEdgeFromATP(X)        // E_X is a UC-edge if X is an activation
                                          //   TP, else ⊥
8     if (δ_XC < localInfo.distFrom[X]) then
9        localInfo.distFrom[X] := δ_XC    // Record shorter path-length from X
                                          //   to C
10       if (δ_XC < Δ_C) then             // Back-propagate only if δ_XC < Δ_C
11          if (X ≡ C) then               // Case 1: Found a loop from C back to C
12             if (δ_XC < 0) then return ⊥    // Negative Loop in LO-graph:
                                              //   STNU not DC
13             else localInfo.CC_loop? := ⊤   // Will trigger separate forward
                                              //   propagation
14          else if ((E_X != ⊥) and (status[E_X] == unstarted)) then
               // Case 2: encountered an unstarted UC-edge; will (later) trigger
               //    recursive processing of E_X
15             localInfo.unstarted-UCEs.add((E_X, X))
16          else if ((E_X != ⊥) and (status[E_X] == started)) then
               // Case 3
17             return ⊥                       // Cycle of recursive calls
18          else              // Case 4: Back-propagate from X along edges in LO-graph
19             foreach
                ((W, δ_WC) ∈ NewApplyRelaxLower(G, X, Δ_C, δ_XC)) do
20                if (δ_WC < G.getOrdEdgeWt(W, C)) then   // If no edge,
                                                          //   ordEdgeWt = ∞
21                   newKey := δ_WC + h(W)
22                   Q.insertOrDecreaseKeyIfSmaller(W, newKey)
```

Algorithm 13: `fwdPropNotDC`

Input: $\mathcal{G} = (\mathcal{T}_X \cup \mathcal{T}_C, \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, an STNU graph; $C \in \mathcal{T}_C$, related to (A, x, y, C); $\Delta_C = y - x$; `distFrom`, vector of distances from X to C computed by `OneStepBackProp`; h, a potential function for $\mathcal{G}_{\ell o}$

Output: $\top$, iff $(A, c{:}x, C) \in \mathcal{T}_\ell$ can be reduced away along paths in the LO-graph $\mathcal{G}_{\ell o}$ whose nodes X satisfy $\mathtt{distFrom}[X] < \Delta_C$.

1 $\mathcal{Q}$:= a new empty priority queue // Key for each X is adjusted distance, $dist(C, X) - h(C)$

2 $\mathcal{Q}.insert(C, -h(C))$ // Initialize queue for forward propagation from C

3 **while** *(not $\mathcal{Q}.empty()$)* **do**

4 $\quad (X, key(X)) := \mathcal{Q}.extractMinNode()$

5 $\quad dist(C, X) := key(X) + h(X)$ // Real distance from C to X

6 $\quad$ **if** $(\mathtt{distFrom}[X] < \Delta_C)$ **then** // Only consider X if distance from X to C less than Δ_C

7 $\quad\quad$ **if** $(dist(C, X) < 0)$ **then return** $\top$ // The path from C to X *can* reduce-away the LC-edge

8 $\quad\quad$ **foreach** $((X, \delta_{XY}, Y) \in \mathcal{E}_\ell \cup \mathcal{E}_o)$ **do** // All edges from X ignoring lower-case label if present

$\quad\quad\quad$ // See whether key for Y needs to be updated in queue

9 $\quad\quad\quad$ `newKey :=` $dist(C, X) + \delta_{XY} - h(Y)$

10 $\quad\quad\quad \mathcal{Q}.insertOrDecreaseKeyIfSmaller(Y, \mathtt{newKey})$

11 **return** $\bot$ // Was unable to reduce-away the LC-edge

Algorithm 14: The `NewApplyRelaxLower` algorithm

Input: $\mathcal{G} = (\mathcal{T}_X \cup \mathcal{T}_C, \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, an STNU graph; $V \in \mathcal{T}_X$, Δ_C; δ_{VC}

Output: A list of pairs, (W, δ_{WC}), obtained by applying the Relax$^-$ and Lower$^-$ rules to all LO-edges incoming to V, together with the edge (V, δ_{VC}, C).

1 `edges` := $\{\}$

2 **if** $(\delta_{VC} \geq \Delta_C)$ **then return** $\{\}$ // The Relax$^-$ and Lower$^-$ rules don't apply

3 **if** $(V \in \mathcal{T}_C)$ **then**

4 $\quad$ `edges`$.add((A_V, x_V + \delta_{VC}))$ // Apply Lower$^-$ rule to $(A_V, v{:}x_V, V)$ and (V, δ_{VC}, C)

5 **else**

6 $\quad$ **foreach** $((W, \delta_{WV}, V) \in \mathcal{E}_o)$ **do** // All ordinary edge going to V

7 $\quad\quad$ `edges`$.add((W, \delta_{WV} + \delta_{VC}))$ // Apply Relax$^-$ rule to (W, δ_{WV}, V) and (V, δ_{VC}, C)

8 **return** `edges`

University of Verona
Department of Computer Science
Strada Le Grazie, 15
I-37134 Verona
Italy

http://www.di.univr.it