# A Faster Algorithm for Finding Negative Cycles in Simple Temporal Networks with Uncertainty

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

**Roberto Posenato**
University of Verona, Italy

**Abstract**

Temporal constraint networks are data structures for representing and reasoning about time (e.g., temporal constraints among actions in a plan). Finding and computing negative cycles in temporal networks is important for planning and scheduling applications since it is the first step toward resolving inconsistent networks. For Simple Temporal Networks (STNs), the problem reduces to finding *simple* negative cycles (i.e., no repeat nodes), resulting in numerous efficient algorithms. For Simple Temporal Networks with Uncertainty (STNUs), which accommodate actions with uncertain durations, the situation is more complex because the characteristic of a *non-dynamically controllable* (non-DC) network is a so-called *semi-reducible negative* (SRN) cycle, which can have repeat edges and, in the worst case, an exponential number of occurrences of such edges. Algorithms for computing SRN cycles in non-DC STNUs that have been presented so far are based on older, less efficient DC-checking algorithms. In addition, the issue of repeated edges has either been ignored or given scant attention. This paper presents a new, faster algorithm for identifying SRN cycles in non-DC STNUs. Its worst-case time complexity is $O(mn + k^2n + kn\log n)$, where $n$ is the number of timepoints, $m$ is the number of constraints, and $k$ is the number of actions with uncertain durations. This complexity is the same as that of the fastest DC-checking algorithm for STNUs. It avoids an exponential blow-up by efficiently dealing with repeated structures and outputting a compact representation of the SRN cycle it finds. The space required to compactly store accumulated path information while avoiding redundant storage of repeated edges is $O(mk + k^2n)$. An empirical evaluation demonstrates the effectiveness of the new algorithm on an existing benchmark.

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

**Keywords and phrases** Temporal constraint networks, overconstrained networks, negative cycles

**Digital Object Identifier** 10.4230/LIPIcs.TIME.2024.9

**Supplementary Material** *Software (Source code)*: `https://profs.scienze.univr.it/~posenato/software/cstnu/` [17]

## 1 Introduction

A Simple Temporal Network with Uncertainty (STNU) is a data structure for representing and reasoning about time [14]. STNUs are attractive for planning and scheduling applications because they accommodate not only a wide variety of temporal constraints (e.g., duration constraints, deadlines, and inter-action constraints), but also actions with uncertain durations (e.g., taxi rides or battery-charging actions) [5, 15, 6, 11, 18]. In STNUs, actions with uncertain durations are represented by *contingent links.* Each STNU has a graphical form where nodes represent timepoints; labeled, directed edges represent temporal constraints; and additional edges (called LC and UC edges) represent bounds on uncontrollable action durations.

The most important property of an STNU is called *dynamic controllability* (DC). An STNU is DC if there exists a dynamic strategy for executing its controllable timepoints that guarantees that all relevant constraints will be satisfied no matter how the uncertain durations turn out – within their specified bounds. There are many polynomial-time algorithms, called DC-checking algorithms, for determining whether any given STNU is DC. The fastest is the $O(mn + k^2n + kn \log n)$-time RUL$^-$ algorithm due to Cairo *et al.* [3], where $n$ is the number of timepoints; $m$, the number of constraints; and $k$, the number of contingent links. Hunsberger and Posenato subsequently presented a modification of RUL$^-$, called RUL2021, that has the same worst-case complexity, but is an order of magnitude faster in practice [10].

The characteristic feature of a non-DC STNU is that it must contain a *semi-reducible negative* (SRN) cycle [12]. In general, any path from $X$ to $Y$ in an STNU graph is semi-reducible if it entails a path of the same length from $X$ to $Y$ that contains no LC edges. Such entailments can be discovered by generating new edges using constraint-propagation (equivalently, edge-generation) rules. Although finding negative cycles in Simple Temporal Networks (STNs) reduces to finding *simple* negative cycles (i.e., no repeat nodes), finding SRN cycles in STNUs is more complex, given that even *indivisible* SRN cycles in a non-DC STNU can have repeat edges and, in the worst case, an *exponential* number of such edges [9]. (An SRN cycle is indivisible if each proper sub-cycle is non-negative or non-semi-reducible.)

When given a *non*-DC STNU, DC-checking algorithms simply report that the network is not DC; they do not produce an SRN cycle [12, 13, 3, 10]. For applications, it is important to identify SRN cycles so that they can be resolved (e.g., by accepting the cost of weakening constraints or tightening uncertain durations). Existing algorithms for finding SRN cycles in non-DC STNUs [22, 23, 21, 1, 2] are based on older, less efficient DC-checking algorithms; and the issue of repeated edges has been ignored or given scant attention. This paper presents a new, faster algorithm for computing SRN cycles in non-DC STNUs while also rigorously addressing the compact representation of SRN cycles having a large number of repeated edges. The new algorithm modifies the RUL2021 algorithm to accumulate path information without impacting its time complexity. The additional space required to compactly store path information, while avoiding redundant storage of repeated edges, is $O(mk + k^2n)$.

## 2 Background

This section summarizes the basic definitions and results for STNUs and then describes the RUL2021 DC-checking algorithm that is the starting point for our new algorithm.

### 2.1 Simple Temporal Networks with Uncertainty

A Simple Temporal Network with Uncertainty (STNU) is a triple $(\mathcal{T}, \mathcal{C}, \mathcal{L})$ where $\mathcal{T}$ is a set of $n$ real-valued variables; $\mathcal{C}$ is a set of $m$ binary difference constraints, each of the form $Y - X \leq \delta$, where $X, Y \in \mathcal{T}$ and $\delta \in \mathbb{R}$; and $\mathcal{L}$ is a set of $k$ *contingent links*, each of the form $(A, x, y, C)$, where $A, C \in \mathcal{T}$ and $0 < x < y < \infty$ [14]. The timepoints typically represent starting or ending times of actions; the constraints can represent deadlines, release times, and duration or inter-action constraints. The contingent links represent actions with *uncertain* durations. For each contingent link $(A, x, y, C)$, $A$ is called the *activation* timepoint and $C$ the *contingent* timepoint. We let $\Delta_C = y - x$. The executor of the network typically controls $A$, but not $C$. The executor only *observes* the execution of $C$ in real-time, knowing only that $C$ will be executed such that $C - A \in [x, y]$. For example, your taxi ride might be represented by the contingent link $(A, 15, 25, C)$, where $A$ is when you enter the taxi, $C$ is when you arrive at your destination, and $C - A \in [15, 25]$ is the uncertain duration, learned only when you arrive.

Each STNU has a graph $(\mathcal{T}, \mathcal{E})$ where the timepoints serve as nodes and the constraints in $\mathcal{C}$ and the contingent links in $\mathcal{L}$ correspond to different kinds of labeled, directed edges. For convenience, edges such as $X \xrightarrow{\alpha} Y$ will be notated as $(X, \alpha, Y)$, where $X, Y \in \mathcal{T}$ and $\alpha \in \mathbb{R}$, possibly annotated with an alphabetic letter. In particular, $\mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u$, where: $\mathcal{E}_o = \{(X, \delta, Y) \mid (Y - X) \leq \delta \in \mathcal{C}\}$ is the set of *ordinary* edges; $\mathcal{E}_\ell = \{(A, c{:}x, C) \mid (A, x, y, C) \in \mathcal{L}\}$, the set of lower-case (LC) edges; and $\mathcal{E}_u = \{(C, C{:}{-y}, A) \mid (A, x, y, C) \in \mathcal{L}\}$, the set of upper-case (UC) edges. Note that each contingent link has a corresponding pair of edges: an LC edge representing that the contingent duration might take on its minimum value $x$, and a UC edge representing that it might take on its maximum value $y$.

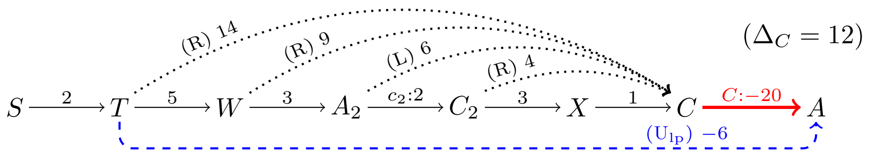
$\mathcal{T}_c$ denotes the set of contingent timepoints; and $\mathcal{T}_x = \mathcal{T} \backslash \mathcal{T}_c$ the set of executable (or controllable) timepoints. An STNU is *dynamically controllable* (DC) if there exists a dynamic strategy for executing its controllable timepoints such that all constraints in $\mathcal{C}$ will necessarily be satisfied no matter how the contingent durations turn out within their specified bounds [14, 7]. An execution strategy is *dynamic* if it can react, in real-time, to observations of contingent executions. The RUL$^-$ algorithm [3] is the DC-checking algorithm with the best worst-case time complexity: $O(mn + k^2 n + kn \log n)$. However, RUL2021, which is a modification of RUL$^-$, has been shown to be an order-of-magnitude faster on a variety of STNU benchmarks, although having the same theoretical complexity [10].

## 2.2 The RUL2021 DC-Checking algorithm

This section summarizes important features of the RUL2021 algorithm. Like all DC-checking algorithms, it operates on the STNU graph, using edge-generation rules to generate new edges representing constraints that must be satisfied by any dynamic execution strategy. Table 1 shows the edge-generation rules used by RUL2021. The R and L rules (for *Relax* and *Lower Case*, respectively) are used to back-propagate distance information in the LO-graph (i.e., the subgraph comprising the LC and ordinary edges). The wavy arrows represent paths in the LO-graph that have already been explored. In the R rule, back-propagation continues along the ordinary edge $(P, v, Q)$ to generate the distance information represented by the dotted edge $(P, v + w, C_i)$. In the L rule, back-propagation continues along the LC edge $(A_j, c_j{:}x_j, C_j)$ to generate the distance information represented by the dotted edge $(A_j, x_j + w, C_i)$. RUL2021 uses the L and R rules only to accumulate distance information; the dotted edges are *not* inserted into the STNU graph. But RUL2021 *does* insert the edges generated by the U$_{\text{lp}}$ rule: ordinary edges that effectively *bypass* UC edges. For example, in the table, the wavy path $(P, v, C_i)$ represents distance information previously generated by the R and L rules. This "edge" combines with the UC edge $(C_i, C_i{:}{-y_i}, A_i)$ to generate the (blue and dashed) bypass edge $(P, v - y_i, A_i)$.

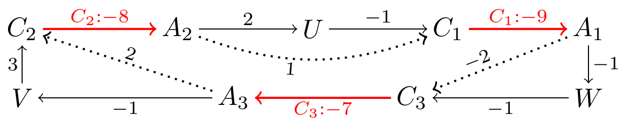
**Table 1** The edge-generation rules for the RUL2021 algorithm.

| Rule | Graphical representation | Applicability Conditions |
|---|---|---|
| R | $\begin{tikzcd} P \arrow[r, "v"] \arrow[rr, dotted, bend right, "v+w"'] & Q \arrow[r, squiggly, "w"] & C_i \end{tikzcd}$ | $Q \in \mathcal{T}_X, w < \Delta_{C_i}, C_i \in \mathcal{T}_C$ |
| L | $\begin{tikzcd} A_j \arrow[r, "c_j{:}x_j"] \arrow[rr, dotted, bend right, "x_j+w"'] & C_j \arrow[r, squiggly, "w"] & C_i \end{tikzcd}$ | $C_j \not\equiv C_i, w < \Delta_{C_i}, C_i \in \mathcal{T}_C$ |
| U$_{\text{lp}}$ | $\begin{tikzcd} P \arrow[r, squiggly, "v"] \arrow[rr, dashed, bend right, "v-y_i"'] & C_i \arrow[r, "C_i{:}{-y_i}"] & A_i \end{tikzcd}$ | $(A_i, x_i, y_i, C_i) \in \mathcal{L}$, $v \geq \Delta_{C_i}$ |



$(\Delta_C = 12)$
(R) 14
(R) 9
(L) 6
(R) 4
$S$ 2 $T$ 5 $W$ 3 $A_2$ $c_2$:2 $C_2$ 3 $X$ 1 $C$ $C$:−20 $A$
($U_{lp}$) −6

Figure 1 RUL2021 generating a (blue and dashed) bypass edge for a (red) UC edge.



$C_2$ $C_2$:−8 $A_2$ 2 $U$ −1 $C_1$ $C_1$:−9 $A_1$
3 2 1 −2 −1
$V$ −1 $A_3$ $C_3$:−7 $C_3$ −1 $W$

Figure 2 A cycle of interruptions detected by RUL2021.

Figure 1 shows how RUL2021 processes a (red) UC edge, assuming that $\Delta_C = 12$. First, it uses the R and L rules to back-propagate from $C$ along LO-edges, collecting distance information indicated by the dotted arrows. (The rules used to generate these edges are in parentheses.) Back-propagation continues as long as the distance stays less than $\Delta_C$. Since the path from $T$ to $C$ has length $14 \geq \Delta_C$, back-propagation stops. Then the $\text{U}_{\text{lp}}$ rule is applied to $(T, 14, C)$ and $(C, C{:}{-20}, A)$ to generate the (dashed) bypass edge $(T, -6, A)$.

There can be many paths emanating backward from $C$ in the LO-graph. To ensure that only *shortest* distances are accumulated, back-propagation is guided by a priority queue and a potential function to re-weight the edges to non-negative values, as in Johnson's algorithm [4], except that here the potential function is a solution to the LO-graph, viewed as an STN. The potential function is initialized by a one-time call to the Bellman-Ford algorithm [4].

Once all of the bypass edges are computed for a given UC edge, they are inserted into the LO-graph, which typically requires updating the potential function. Since all of the new edges terminate at $A$, this updating can be carried out by a separate Dijkstra-like back-propagation from $A$ using a priority queue and the pre-existing potential function. If all of the UC edges can be successfully processed in this way, then RUL2021 declares the STNU to be DC.

However, three kinds of events can signal that the STNU is not DC:

1. **Failure to update the potential function.** Inserting new bypass edges might cause the LO-graph to become inconsistent (as an STN), which would be detected by encountering a negative cycle in the LO-graph while trying to update the potential function.
2. **Cycle of interruptions.** When processing a UC edge $\mathbf{E}_1$, back propagation from its contingent timepoint $C_1$ might bump into a different UC edge $\mathbf{E}_2$. If so, RUL2021 *interrupts* its processing of $\mathbf{E}_1$ to process $\mathbf{E}_2$. After finishing with $\mathbf{E}_2$, back propagation from $\mathbf{E}_1$ continues. However, should a *cycle* of such interruptions occur, for example, as illustrated in Figure 2, then the network cannot be DC. In the figure, the UC edges are colored red; distances along LO-paths computed by back-propagation using the L and R rules are indicated by dotted arrows; and the relevant contingent links are $(A_1, 1, 9, C_1)$, $(A_2, 2, 8, C_2)$ and $(A_3, 3, 7, C_3)$. Now back propagation from $C_1$ should continue as long as the dotted distances are less than $\Delta_{C_1} = 9 - 1 = 8$, but is interrupted by the UC edge $(C_2, C_2{:}{-8}, A_2)$. Similarly, back-propagation from $C_2$ should continue as long as the dotted distances are less than $\Delta_{C_2} = 8 - 2 = 6$, but is interrupted by the UC edge $(C_3, C_3{:}{-7}, A_3)$. Finally, back propagation from $C_3$ should continue as long as the dotted distances are less than $\Delta_{C_3} = 7 - 3 = 4$, but is interrupted by the first UC edge,

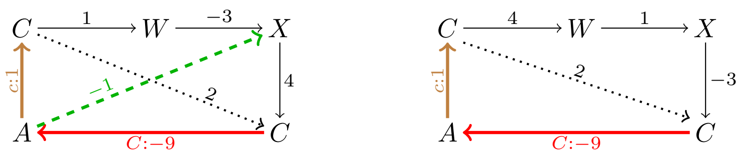


**Figure 3** Two different CC loops associated with a contingent link $(A, 1, 9, C)$.

**Algorithm 1** The `FindSRNC` algorithm.

```
   Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E} = \mathcal{E}_o \cup \mathcal{E}_\ell \cup \mathcal{E}_u)$, an STNU graph
   Output: $(negCycle, edgeAnn)$, where $negCycle$ is an SRN cycle and $edgeAnn$ is a hash table of
           path annotations for edges in the cycle; or $(\emptyset, \emptyset)$ if the STNU is DC
 1 glo := new global data structure                    // Fields: pf, status, intBy, edgeAnnotation
 2 glo.pf := BellmanFord($\mathcal{G}_{\ell o}$)                      // Initialize potential function for LO-graph
 3 if glo.pf == $\perp$ then return BFCT($\mathcal{G}_{\ell o}$)
 4 glo.edgeAnnotation := new empty hash table
 5 glo.status := [nYet, ..., nYet]                 // Initial processing status of the $k$ UC edges
 6 glo.intBy := [$\perp$, ..., $\perp$]                                  // $k$-vector: records interruptions
 7 foreach $(C, C{:}{-y}, A) \in \mathcal{E}_u$ do
 8     $negCycle$ := RulBackProp($\mathcal{G}$, $(C, C{:}{-y}, A)$, glo)
 9     if $negCycle \neq \emptyset$ then return $(negCycle$, glo.edgeAnnotation$)$
10 return $(\emptyset, \emptyset)$
```

thereby completing the cycle. At this point, RUL2021 signals that the STNU is not DC. This is justified since each dotted distance being less than the corresponding $\Delta_{C_i}$ value ensures that the length of the cycle is negative; and a negative cycle in the OU-graph (i.e., the subgraph comprising ordinary and UC edges) represents an impossible-to-satisfy constraint for a dynamic execution strategy.

3. **CC loops.** Back propagation from a UC edge $(C, C{:}{-y}, A)$ can also be blocked if an LO-path from $C$ back to $C$ of length less than $\Delta_C$ is encountered. Such a path is called a *CC loop* [10]. A CC loop does not necessarily imply that the STNU is not DC; but it sometimes does. Figure 3 illustrates two scenarios in which back-propagation from $C$ reveals a CC loop of length $2 < 8 = \Delta_C$. However, the lefthand STNU is not DC, while the righthand one is. The key difference, according to Morris' analysis of semi-reducible paths [12], is that the lefthand graph contains a *negative* LO-path emanating from $C$ (to $X$) which can be used to generate the (dashed, green) bypass edge $(A, -1, X)$, thereby creating a negative cycle in the OU-graph from $A$ to $X$ to $C$ to $A$, whereas the righthand graph has no such path.

## 3 The `FindSRNC` (Find Semi-Reducible Negative Cycle) Algorithm

This section introduces our new `FindSRNC` algorithm, which modifies RUL2021 to efficiently accumulate path information. To contrast `FindSRNC` and RUL2021, we have preserved the general structure of RUL2021, although to improve readability we have expanded the rather cryptic names of the original helper algorithms. Modifications are highlighted in green.

The pseudocode for `FindSRNC` is in Algorithm 1. When given a non-DC STNU as input, it outputs a compact representation of an SRN cycle in the form $(negCycle, edgeAnn)$, where $negCycle$ is a negative cycle of edges in the LO- or OU-graph, depending on how the cycle arose; and $edgeAnn$ is a hash table of $(key, value)$ pairs, where each $key$ identifies an (ordinary) bypass edge generated by the algorithm, and *value* is the path used to generate that edge. It is efficient to present the SRN cycle in this way since, in the worst case, unpacking all of the edges in the cycle could result in an exponential number of repeated edges.

Like RUL2021, `FindSRNC` starts by calling the Bellman-Ford algorithm to create an initial potential function for the LO-graph which, if successful, is stored in the `pf` field of a `glo` data structure. (The `glo` data structure contains **glo**bal information accessible across multiple recursive calls to process UC edges.) If Bellman-Ford fails, then `FindSRNC` calls the $O(mn)$-time BFCT algorithm [19] to return a negative cycle for the LO-graph (an STN). A negative cycle in the LO-graph is a trivial case of an SRN cycle for an STNU.

If Bellman-Ford succeeds, `FindSRNC` initializes `glo.edgeAnnotation` to a new hash table that will record the paths from which any bypass edges are derived. The `glo.status` field tracks the processing status of each UC edge, as in RUL2021. The `glo.intBy` field, initially a vector of $\perp$ entries, stores information about when the processing of one UC edge is interrupted by another. Finally, `FindSRNC` iterates through the UC edges, processing each with a call to `RulBackProp` (Algorithm 2). Because `RulBackProp` recursively processes any interrupting UC edges, by the time `FindSRNC` calls `RulBackProp` on some UC edge, it may have already been processed. The `status` field is used to avoid redundant processing.

## `RulBackProp`

The pseudocode for the `RulBackProp` algorithm is in Algorithm 2. It processes a single UC edge $\mathbf{E} = (C, C{:}{-}y, A)$ while integrating the recursive processing of any interrupting UC edges. At Line 2, if $\mathbf{E}$ has already been processed, it immediately returns $\top$. At Line 3, it checks whether the processing of $\mathbf{E}$ has already been started, but not yet completed, which implies a negative cycle of interruptions. In this case, `RulBackProp` calls `AccNegCycle` (Algorithm 3) to collect the relevant path information accumulated in the `glo.intBy` vector, which is then returned as a compact representation of an SRN cycle. (More will be said about how the information in `glo.intBy` is generated.)

In the pseudocode, we use `+` as a concatenation operator that can be applied to edges or paths. For example, if $e_1$ is an edge, and $\pi_1$ and $\pi_2$ are paths, then $\pi_1 + e_1 + \pi_2$ represents their concatenation into a single path. In addition, we use $\langle\rangle$ to denote the empty path.

At Lines 4–7, `RulBackProp` prepares to process a UC edge $\mathbf{E} = (C, C{:}{-}y, A)$. As in RUL2021, `ccLoop` is a flag used to signal the discovery of a *CC loop*; and `dist` records, for each encountered timepoint $X$, the distance from $X$ to $C$ in the LO-graph. A new field, `path`, records the paths from each $X$ to $A$ (via $C$). Back-propagation from $C$ is governed by a priority queue $\mathcal{Q}$, initialized at Lines 8–10 to include each $X$ connected to $C$ by an edge.

In each iteration of the `while` loop (Lines 12–23), `RulBackProp` either starts *or resumes* the processing of $\mathbf{E}$, first (at Line 13) by calling `TryBackProp` (Algorithm 4). `TryBackProp` (described later) back-propagates along LO-edges, but does *not* generate or insert any bypass edges. Instead, it simply collects the relevant distance *and path* information, while also keeping track of whether it encountered any unstarted (i.e., interrupting) UC edges or *CC loops.* At Line 14, `RulBackProp` checks whether `TryBackProp` found an SRN cycle, in which case `RulBackProp` returns that cycle. Otherwise, at Line 15, `RulBackProp` checks whether `TryBackProp` encountered any interrupting UC edges. If so, for each interrupting UC edge $\mathbf{E}_X$ (Lines 16–19), it uses `glo.intBy` $[\mathbf{E}]$ to record the interruption and then attempts to recursively process $\mathbf{E}_X$. If all interrupting UC edges are successfully processed, it clears the `glo.intBy` $[\mathbf{E}]$ entry (Line 20) and prepares for the next iteration of the `while` loop by re-initializing the priority queue (Lines 21–22) so that processing $\mathbf{E}$ can be resumed, starting from the activation timepoints of the no-longer-interrupting UC edges.

**Algorithm 2** The `RulBackProp` algorithm.

```
   Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, STNU graph; $\mathbf{E} = (C, C{:}{-y}, A) \in \mathcal{E}_u$, a UC edge; glo, a global structure
   Output: $negCycle$, an SRN cycle; or $\emptyset$ if $\mathbf{E}$ successfully processed
 1 $h :=$ glo.pf                                              // Potential function for LO-graph
 2 if $glo.status[\mathbf{E}] ==$ done then return $\top$             // $\mathbf{E}$ already done
 3 if $glo.status[\mathbf{E}] ==$ started then return $AccNegCycle(glo.intBy, \mathbf{E})$   // Cycle of interrupts
 4 glo.status[$\mathbf{E}$] := started                    // Prepare to start processing the UC edge $\mathbf{E}$
 5 loc := new local struct; loc.ccLoop := $\bot$                      // No CC loop found yet
 6 loc.dist := $[\infty, \dots, \infty]$                              // distance from each TP to $C$
 7 loc.path := $[\langle\rangle, \dots, \langle\rangle]$                        // path from each TP to $A$ (via $\mathbf{E}$)
 8 $\mathcal{Q}$ := a new priority queue   // Priority of each $X$ is $h(X) + \delta_{xc}$, adjusted dist. from X to C
 9 foreach $(X, \delta_{xc}, C) \in \mathcal{E}_o$ do
10     $\mathcal{Q}.ins(X, h(X) + \delta_{xc})$; loc.path[$X$] := $(X, \delta_{xc}, C) + (C, C{:}{-y}, A)$
11 $continue? := \top$
12 while $continue?$ do                          // Start or resume processing of UC edge $\mathbf{E}$
13     $negCycle$ := TryBackProp($\mathcal{G}, \mathbf{E}, \mathcal{Q}$, glo, loc)
14     if $negCycle \neq \emptyset$ then return $negCycle$
15     if loc.UnstartedUCs $\neq \emptyset$ then                     // Process unstarted UC-edges
16         foreach $(\mathbf{E}_X, X) \in$ loc.UnstartedUCs do
17             glo.intBy[$\mathbf{E}$] := ($\mathbf{E}_X$, loc.path[$X$])
18             $negCycle$ := RulBackProp($\mathcal{G}, \mathbf{E}_X$, glo)
19             if $negCycle \neq \emptyset$ then return $negCycle$
20         glo.intBy[$\mathbf{E}$] := $\bot$                          // All interruptions of $\mathbf{E}$ completed
21         $\mathcal{Q}.clear()$                                // Prepare $\mathcal{Q}$ for next iteration of WHILE
22         foreach $(\mathbf{E}_X, X) \in$ loc.UnstartedUCs do $\mathcal{Q}.ins(X$, loc.dist[$X$] + glo.pf[$X$])
23     else $continue? := \bot$                                  // Back-prop. from $\mathbf{E}$ completed
24 if loc.ccLoop then                     // CC-loop found; must initiate forward propagation
25     $(X, \mathcal{P}_X)$ := FwdPropNDC($\mathcal{G}, C, \Delta_C$, loc, glo.pf)      // $\Delta_C = y - x$ for cont. link $(A, x, y, C)$
26     // If $(A, c{:}x, C)$ can be reduced away, then return SRN cycle
27     if $(X, \mathcal{P}_X) \neq \emptyset$ then return $(A, c{:}x, C) + \mathcal{P}_X + loc.path[X]$
28 foreach $X \in \mathcal{T} \setminus \{C\}$ do                         // Generate bypass edges using $\mathrm{U_{lp}}$ rule
29     $\delta_{xc}$ := loc.dist[$X$]                               // $\delta_{xc} = \infty$ means node not reachable
30     if $\Delta_C \leq \delta_{xc} < \infty$ then
31         $\mathcal{G}.insertOrdEdge(X, \delta_{xc} - y, A)$
32         glo.edgeAnnotation.$put((X, A)$, loc.path[$X$])
33         $edges? := \top$
34 if $edges?$ then (glo.pf, $negCycle$) := UpdatePotFn($\mathcal{G}, A$, glo.pf)
35 if glo.pf $== \bot$ then return $negCycle$
36 glo.status[$\mathbf{E}$] := done
37 return $\emptyset$                                      // Processing of $\mathbf{E}$ successfully completed
```

Once all back-propagation from $\mathbf{E}$ is done, `RulBackProp` checks, at Line 24, whether any *CC loops* were encountered. If so, it calls `FwdPropNDC` to carry out a separate forward propagation from $C$ along LO-edges, checking whether any LO-path, $\mathcal{P}_{CX}$, from $C$ to some $X$, can be used to bypass the LC edge $e = (A, c{:}x, C)$. If so, there must be an SRN cycle, $e + \mathcal{P}_{CX} + \texttt{loc.path}[X]$, where $\texttt{loc.path}[X]$ is the LO-path from $X$ to $A$ obtained by the earlier back-propagation from $C$ [10]. Hence, `FwdPropNDC` returns $(X, \mathcal{P}_{CX})$. For the STNU on the left of Figure 3, $\mathcal{P}_{CX}$ is $(C, 1, W) + (W, -3, X)$ and $\texttt{loc.path}[X]$ is $(X, 4, C) + (C, C{:}{-9}, A)$.

**Algorithm 3** The `AccNegCycle` algorithm (new).

```
Input: glo.intBy, vector recording a cycle of interruptions; $\mathbf{E} \in \mathcal{E}_u$, a UC edge in the cycle
Output: $negCycle$, an SRN cycle containing $\mathbf{E}$
1 $negCycle := \langle\rangle$
2 $(\mathbf{E}', P') :=$ glo.intBy$[\mathbf{E}]$                // $\mathcal{P}'$ is path used to generate $\mathbf{E}'$
3 while $\mathbf{E}' \neq \mathbf{E}$ do
4     $negCycle := P' + negCycle$                            // Accumulate $\mathcal{P}'$ into cycle
5     $(\mathbf{E}', P') :=$ glo.intBy$[\mathbf{E}']$                    // Fetch next interrupter
6 return $P' + negCycle$
```

If forward propagation fails to find an SRN cycle, then `RulBackProp` finally uses the information in `loc.dist` to generate edges that bypass the UC edge **E** (Lines 28–33). These are the only edges that `FindSRNC` actually inserts into the STNU. For each bypass edge $(X, \delta_{xc} - y, A)$, the corresponding path that has been accumulated in `loc.path`$[X]$ is recorded in the `glo.edgeAnnotation` hash table (Line 32). (As discussed below, it is `TryBackProp` that accumulates the path information in `loc.path`$[X]$.) If any bypass edges are inserted, then `RulBackProp` (at Line 34) calls `UpdatePotFn` (Algorithm 7, discussed later) to update the potential function for the LO-graph, whence the processing of **E** is completed (Line 36).

## `TryBackProp`

Pseudocode for `TryBackProp` (called `phaseOne` in RUL2021) is given as Algorithm 4. For a UC edge $\mathbf{E} = (C, C{:}{-y}, A)$, it propagates *backward* from $C$ along LO-edges as long as the accumulated distance remains less than $\Delta_C = y - x$. (Recall the condition $w < \Delta_{C_i}$ for the R and L rules in Table 1.) Its `while` loop (Lines 3–19) uses the priority queue initialized by `RulBackProp` and the potential function updated by `RulBackProp` to explore shortest paths in the LO-graph. At Lines 4–6, it pops a node $X$ off the queue, converts its key into the distance from $X$ to $C$, and assigns it to `loc.dist`$[X]$. At Line 7, it checks whether $X$ is an activation timepoint and, if so, sets $\mathbf{E}_X$ to the corresponding UC edge. Next, if `loc.dist`$[X] < \Delta_C$ (Line 8), `TryBackProp` considers four cases (Lines 9-19).
In Case 1, back propagation has circled back to $C$, prompting `TryBackProp` to set the `ccLoop` flag. In Case 2, back propagation has encountered another UC edge $\mathbf{E}_X$ whose processing has not yet been started; so `TryBackProp` pushes the interrupting edge onto a list of as-yet-unstarted UC edges (to be processed later by `RulBackProp`). In Case 3, back propagation has hit a UC edge $\mathbf{E}_X$ whose processing has already been started, but not yet finished. This implies a cycle of interruptions and, hence, an SRN cycle. In preparation for terminating, the algorithm records the interruption of **E** by $\mathbf{E}_X$, along with the *path* from $X$ to $A$ accumulated in `loc.path`$[X]$. Then it calls `AccNegCycle` (Algorithm 3) to recursively collect the information accumulated in the cycle of interruptions to return an SRN cycle.
In Case 4, back propagation continues past $X$, and path information is accumulated. At Line 15, `TryBackProp` calls `ApplyRL` (Algorithm 5), which applies the R rule to all ordinary edges coming into $X$ and, if $X$ happens to be a contingent timepoint, applies the L rule to the corresponding LC edge coming into $X$. `ApplyRL` returns a list of pairs, each of the form $(e, \delta_{WC})$, where $e$ is an LO-edge from $W$ to $X$, and $\delta_{WC}$ is the length of the LO-path from $W$ to $C$ via $X$. For each such pair, `TryBackProp` (at Lines 15–19) first checks whether the path from $W$ to $C$ represents a new shorter LO-path and, if so, updates the key for $W$ in the priority queue and incrementally accumulates the relevant path information in `loc.path`$[W]$.

**Algorithm 4** The `TryBackProp` algorithm.

```
Input: $\mathcal{G} = (\mathcal{T}, \mathcal{E})$, an STNU graph; $\mathbf{E} = (C, C{:}{-y}, A) \in \mathcal{E}_u$; $\mathcal{Q}$, a priority queue; glo, global
        struct; loc, local struct
Output: $negCycle$, an SRN cycle; or $\emptyset$ if no SRN cycle found.
1  $h :=$ glo.pf                                   // Potential function, a solution to the LO-graph
2  loc.UnstartedUCs $:= \{\}$                        // Will collect unstarted UC edges
3  while $\mathcal{Q} \neq \emptyset$ do
       // $key_X$ = distance from $X$ to $C$, adjusted by $h$
4      $(X, key_X) := \mathcal{Q}.extractMinNode()$
5      $\delta_{xc} := key_X - h(X)$                      // $\delta_{xc}$ = distance from $X$ to $C$ in $\mathcal{G}_{\ell o}$
6      loc.dist$[X] := \delta_{xc}$                       // Record shorter length
       // If $X$ is an ATP, then $\mathbf{E}_X$ is corresponding UC-edge; else $\perp$
7      $\mathbf{E}_X := \mathcal{G}.UCEdgeFromATP(X)$
8      if $\delta_{xc} < \Delta_C$ then                        // Continue back-propagation
           // Case 1: Found CC loop of length $\delta_{xc} < \Delta_C$; signal need for fwd prop
9          if $X \equiv C$ then loc.ccLoop $:= \top$
           // Case 2: $\mathbf{E}_X$ is an unstarted UC-edge; accumulate it
10         else if glo.status$[\mathbf{E}_X]$ == nYet then loc.UnstartedUCs.$add((\mathbf{E}_X, X))$
           // Case 3: Cycle of interruptions: not DC
11         else if glo.status$[\mathbf{E}_X]$ == started then
12             glo.intBy$[\mathbf{E}] := (\mathbf{E}_X,$ loc.path$[X])$
13             return AccNegCycle (glo.intBy, $\mathbf{E}_X$)
14         else // Case 4: Continue back-propagation along LO-edges
15             foreach $(e, \delta_{wc}) \in$ ApplyRL$(\mathcal{G}, X, \Delta_C, \delta_{xc})$ do
16                 $newKey := \delta_{wc} + h(W)$
17                 if $\delta_{wc} <$ loc.dist$[W]$ and $(W \notin \mathcal{Q}$ or $newKey < \mathcal{Q}.key(W))$ then
                       // Accumulate new path from $W$ to $C$
18                     $\mathcal{Q}.insOrDecrKey(W, newKey)$
19                     loc.path$[W] := e+$loc.path$[X]$
20 return $\emptyset$
```

**Algorithm 5** The `ApplyRL` algorithm.

```
Input: $\mathcal{G}$, an STNU graph; $X \in \mathcal{T}$; $\Delta_C$; and $\delta_{xc} < \Delta_C$
Output: A list of pairs, $(e, \delta_{wc})$, where $e$ is an LO-edge from $W$ to $X$, and $\delta_{WC} = |e| + \delta_{xc}$.
1  $edgeDistPairs := \{\}$
   // If $X$ is a contingent timepoint $C_i$, then apply the L rule to $(A_i, c_i{:}x_i, C_i)$ and $(C_i, \delta_{xc}, C)$
2  if $X \equiv C_i \in \mathcal{T}_C$ then $edgeDistPairs.add(((A_i, c_i{:}x_i, C_i), x_i + \delta_{xc}))$
3  else // Otherwise, apply the R rule to $(W, \delta_{wx}, X)$ and $(V, \delta_{vc}, C)$
4      foreach $(W, \delta_{wx}, X) \in \mathcal{E}_o$ do $edgeDistPairs.add(((W, \delta_{wx}, X), \delta_{wx} + \delta_{xc}))$
5  return $edgeDistPairs$
```

## `FwdPropNDC`

The `FwdPropNDC` algorithm (Algorithm 6) propagates *forward* from $C$ along LO-edges checking whether there is a negative-length path from $C$ to some $X$ that can be used to bypass the LC edge $(A, c{:}x, C)$. It is the same as in RUL2021, except that it accumulates path information in a vector called `fwdPath`. At Lines 2–3, a priority queue is initialized to contain just $C$, with `fwdPath`$[C] = \langle\rangle$. The priority queue uses the same potential function as `TryBackProp` to effectively re-weight the LO-edges. As each timepoint $X$ is popped from the queue (Line 5),

**Algorithm 6** The `FwdPropNDC` algorithm.

```
Input: $\mathcal{G}$, an STNU graph; $C \in \mathcal{T}_C$; $\Delta_C = y - x$; loc, local struct; $h$, potential function.
Output: $(X, \mathcal{P}_{CX})$, if path $\mathcal{P}_{CX}$ can be used to reduce away the LC edge $(A, c{:}x, C)$; else $\emptyset$
1  fwdPath := $\{\langle\rangle, \ldots, \langle\rangle\}$          // For each $X$, $fwdPath[X]$ is an LO-path from $C$ to $X$
2  $\mathcal{Q}$ := new priority queue                         // Key $key_X = d(C, X) - h(C)$
3  $\mathcal{Q}.insert(C, -h(C))$                               // Queue initially contains only $C$
4  while $\mathcal{Q} \neq \emptyset$ do
5      $(X, key_X) := \mathcal{Q}.extractMinNode()$
6      $d(C, X) := key_X + h(X)$                                // Distance from $C$ to $X$ in $\mathcal{G}_{\ell o}$
7      if $loc.dist[X] < \Delta_C$ then                         // If distance from $X$ to $C < \Delta_C$
           // Check if the path $CX$ can reduce-away the LC-edge
8          if $d(C, X) < 0$ then return $(X, fwdPath[X])$
9      foreach $(X, \delta_{xy}, Y) \in \mathcal{E}_\ell \cup \mathcal{E}_o$ do       // Iterate over LO-edges emanating from $X$
10         $newKey := d(C, X) + \delta_{xy} - h(Y)$
11         if $Y \notin \mathcal{Q}$ or $newKey < \mathcal{Q}.key(Y)$ then
12             $\mathcal{Q}.insOrDecrKey(Y, newKey)$
13             $fwdPath[Y] := fwdPath[X] + (X, \delta_{xy}, Y)$
14 return $\emptyset$                                          // Was unable to reduce-away the LC-edge
```

the distance from $X$ to $C$ that was determined during back-propagation and stored in `loc.dist`$[X]$ is compared to $\Delta_C$. (Generating an edge to bypass the LC edge using the path from $C$ to $X$ will only create an SRN cycle if `dist`$[X] < \Delta_C$ [10].) If `dist`$[X] < \Delta_C$ and $d(C, X) < 0$ (i.e., an appropriate negative-length path has been found), then `FwdPropNDC` terminates, returning $(X,$ `fwdPath`$[X])$ (Line 8). Otherwise, forward propagation continues from $X$, accumulating relevant path information (Lines 9–13). If the queue is exhausted without finding a way to bypass the LC edge, `FwdPropNDC` returns $\emptyset$ (Line 14).

### `UpdatePotFn`

When `RulBackProp` inserts edges that bypass a UC edge **E**, it changes the LO-graph. Hence, the potential function for the LO-graph typically needs to be updated. The pseudocode for the `UpdatePotFn` function is given as Algorithm 7.

Since all bypass edges for **E** necessarily point at its activation timepoint $A$, `UpdatePotFn` propagates backward from $A$ along LO-edges as long as changes to the potential function, $h$, are required. This function and its helper `UpdateVal` (Algorithm 8) are the same as in RUL2021 except that path information is accumulated (Algorithm 8, Line 6) so that if back-propagation ever cycles all the way back to $A$, the implied SRN cycle can be returned (Algorithm 8, Line 4).

## Computational Complexity

`FindSRNC` performs more operations than RUL2021, mostly by accumulating path information during propagation. For lack of space, we simply note that the most time-consuming operation is prepending an edge onto the front of an existing path, which happens at most once per edge visited. Since the prepending operation $(+)$ can be realized in constant time, the worst-case time complexity of `FindSRNC` is the same as that of RUL2021: $O(mn + k^2n + kn \log n)$.

Regarding the *extra* space requirements of `FindSRNC`, the most costly is the space needed by `TryBackProp` for accumulating path information in the `loc.path` structures. `TryBackProp` is called at most $2k$ times [3, 10]. Each call explores at most $(m + nk)$ edges. (`FindSRNC`

**Algorithm 7** The `UpdatePotFn` algorithm.

```
Input: $\mathcal{G}$, an STNU graph; $A$, an activation timepoint; $h$, a potential function for $\mathcal{G}_{\ell o}$, excluding
       edges ending at $A$
Output: $(h', negCycle)$, where $h'$ is either a potential function for $\mathcal{G}_{\ell o}$ (including edges
        terminating at $A$); or $\perp$, the latter indicating that $negCycle$ is a negative cycle
1  $h'$ := copy of $h$;   $path := [\langle\rangle, \ldots, \langle\rangle]$
2  $\mathcal{Q}$ := new priority queue;   $\mathcal{Q}.insert(A, 0)$        // Initialize queue for back-prop from $A$
3  while $\mathcal{Q} \neq \emptyset$ do
4      $(V, key_V) := \mathcal{Q}.extractMinNode()$
5      foreach $((U, \delta, V) \in \mathcal{E}_o)$ do        // Back-propagate along ordinary edges ending at $V$
6          $negCycle :=$ UpdateVal$((U, \delta, V), h, h', \mathcal{Q}, path)$
7          if $negCycle \neq \emptyset$ then  return $(\perp, negCycle)$
8      if $V \in \mathcal{T}_C$ then        // $V$ is contingent; back-propagate along LC edge $(A_V, v{:}x_V, V)$
9          $negCycle :=$ UpdateVal$((A_V, x_V, V), h, h', \mathcal{Q}, path)$
10         if $negCycle \neq \emptyset$ then  return $(\perp, negCycle)$
11 return $(h', \emptyset)$
```

**Algorithm 8** The `UpdateVal` algorithm.

```
Input: $(U, \delta, V)$, an edge; $h$, $h'$, potential fns.; $\mathcal{Q}$, priority queue; and $path$, a vector of path info
Output: $negCycle$, an SRN cycle; or $\emptyset$ if $h'$ was successfully updated to satisfy $(U, \delta, V)$
1  Side Effect: Modifies $\mathcal{Q}$, $h'$ and $path$
2  if $h'(U) < h'(V) - \delta$ then
3      $h'(U) := h'(V) - \delta$
       // If back propagation has cycled back to $A$, return the cycle
4      if $\mathcal{Q}.state(U)$ == alreadyPopped then  return $(U, \delta, V) + path[V]$
5      $\mathcal{Q}.insOrDecrKey(U, h(U) - h'(U))$
6      $path[U] := (U, \delta, V) + path[V]$
7  return $\emptyset$
```

inserts at most $nk$ edges overall.) Each edge exploration involves prepending an existing path with an edge, which uses only constant space. So the overall space complexity across *all* calls to `TryBackProp` is $O(mk + k^2n)$. Similar remarks apply to `FwdPropNDC` and `UpdatePotFn`.

The `edgeAnnotation` hash table has at most $nk$ entries: one for each bypass edge. Each entry is a *pointer* to a `loc.path` entry. So the total space required is $O(nk)$. The *compact* SRN cycle generated by `AccNegCycle` is the concatenation of at most $k$ paths, each with at most $n$ edges, for a total of at most $nk$ edges, which is dominated by the $O(mk + k^2n)$ space discussed above. This compact cycle, together with the information in the `edgeAnnotation` hash table, avoids redundantly storing repeated structures. In this way, it uses polynomial space to *implicitly* represent a cycle that, if fully expanded, might have exponentially many edges. Similar remarks apply to the cycles returned by `FwdPropNDC` and `UpdatePotFn`.

## Magic Loop Example

Hunsberger [8] identified a family of STNUs in which the only SRN cycle, called a *magic loop,* has an *exponential* number of edges. Since each STNU has at most $n^2 + 2k$ edges, magic loops necessarily contain a large number of repeated edges. In particular, a magic loop *of order k* has $k$ contingent links, but $3(2^k) - 2$ edges. The top of Figure 4 shows an STNU whose (brown) LC edges are $\mathbf{e}_1 = (A_1, c_1{:}1, C_1)$, $\mathbf{e}_2 = (A_2, c_2{:}1, C_2)$, and $\mathbf{e}_3 = (A_3, c_3{:}1, C_3)$; and whose (red)

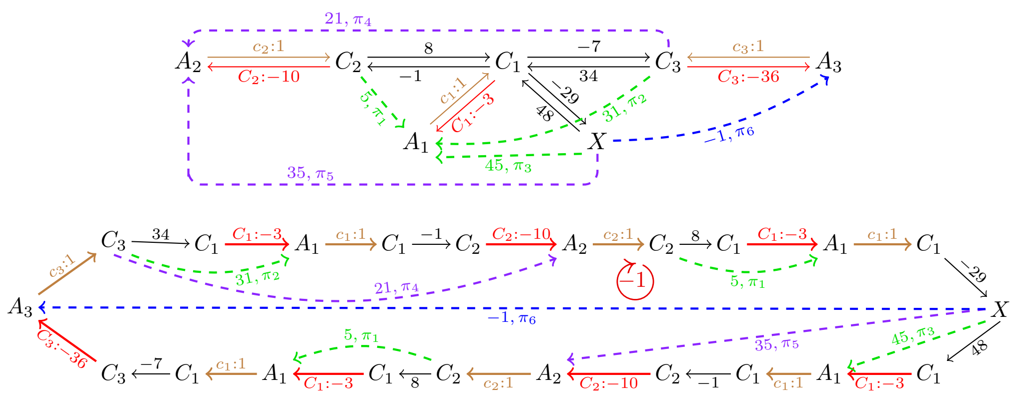
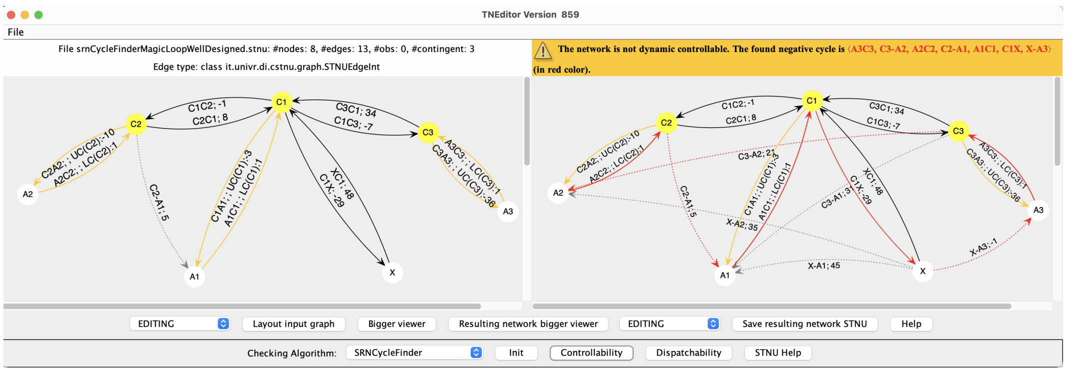


Figure 4 A sample STNU (top) and the magic loop of order 3 (bottom) hiding within it.


TNEditor Version 859

File

File srnCycleFinderMagicLoopWellDesigned.stnu: #nodes: 8, #edges: 13, #obs: 0, #contingent: 3

Edge type: class it.univr.di.cstnu.graph.STNUEdgeInt

The network is not dynamic controllable. The found negative cycle is ⟨A3C3, C3-A2, A2C2, C2-A1, A1C1, C1X, X-A3⟩ (in red color).

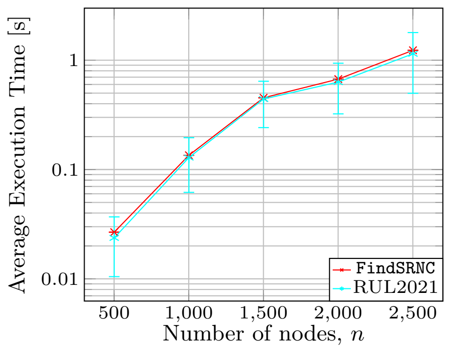
Figure 5 A screenshot of `FindSRNC` in action.

UC edges are $\mathbf{E}_1 = (C_1, C_1{:}-3, A_1)$, $\mathbf{E}_2 = (C_2, C_2{:}-10, A_2)$, and $\mathbf{E}_3 = (C_3, C_3{:}-36, A_3)$. The bypass edges generated by `FindSRNC` are dashed: those bypassing $\mathbf{E}_1$ in green, $\mathbf{E}_2$ in purple, and $\mathbf{E}_3$ in blue. Each bypass edge is also annotated by a path, where: $\pi_1 = (C_2, 8, C_1) + \mathbf{E}_1$; $\pi_2 = (C_3, 34, C_1) + \mathbf{E}_1$; $\pi_3 = (X, 48, C_1) + \mathbf{E}_1$; $\pi_4 = \pi_2 + \mathbf{e}_1 + (C_1, -1, C_2) + \mathbf{E}_2$; $\pi_5 = \pi_3 + \mathbf{e}_1 + (C_1, -1, C_2) + \mathbf{E}_2$; and $\pi_6 = \pi_5 + \mathbf{e}_2 + \pi_1 + \mathbf{e}_1 + (C_1, -7, C_3) + \mathbf{E}_3$. The magic loop for this STNU is at the bottom of the figure. It has 22 edges. $\mathbf{E}_1$ and $\mathbf{e}_1$ appear four times each; several other edges, twice each. After all UC edges have been processed, `UpdatePotFn` finds a negative cycle in the LO-graph: $\pi_6 + \mathbf{e}_3 + \pi_4 + \mathbf{e}_2 + \pi_1 + \mathbf{e}_1 + (C_1, -29, X)$. This information is compactly stored in the cycle returned by `FindSRNC`. For higher-order magic loops, the number of edges grows exponentially, but the space used by `FindSRNC` is bounded by $mk + nk^2$.

## 4 Empirical Evaluation

In this section, we present a possible implementation of the `FindSRNC` algorithm and one its evaluation in a public benchmark.

The proposed algorithm was implemented as a proof-of-concept prototype in the (freely available) CSTNU Tool, version 1.42 [17]. The tool enables users to create different kinds of temporal constraint networks and to verify automatically some properties like dynamic controllability or consistency (for some kinds of networks). In particular, as concerns STNUs, it allows one to verify the dynamic controllability and, in case the network is not DC, to obtain the semireducible negative cycle that determines the non-controllability.



| $n$ | Avg. SRN cycle length | Pct. Simple SRN cycles | Avg. SRN expanded cycle length |
|---|---|---|---|
| 500 | 8.0 | 63% | 14.1 |
| 1000 | 7.9 | 64% | 14.4 |
| 1500 | 7.6 | 74% | 15.3 |
| 2000 | 8.5 | 72% | 14.0 |
| 2500 | 8.6 | 63% | 14.4 |

**Figure 6** Experimental results.

The screenshot Figure 5 shows the CSTNU Tool after the execution of `FindSRNC` algorithm on the STNU depicted in Figure 4. On the left side, there is the initial network that can be edited. On the right side, there is the checked network with the semireducible negative cycle emphasized in red. The status bar above the network on the right gives a summary of the `FindSRNC` result. The extended result (like the expanded semireducible negative cycle) is saved in a logging file associated with the execution.

We empirically evaluated `FindSRNC` on a published benchmark [16] to confirm that the execution times of `FindSRNC` and RUL2021 are equivalent, and to highlight the characteristics of the SRN cycles in non-DC instances. Our implementations are publicly available [17]. We ran them on a JVM 21 with 8 GB of heap memory on a Apple PowerBook/M1 Pro.

For each $n \in \{500, 1000, 1500, 2000, 2500\}$, the benchmark contains 200 randomly generated non-DC STNUs, each having $n$ nodes, $n/10$ contingent links, and $m \approx 3n$ edges. For each sub-benchmark (i.e., for each $n$), we used the first 100 instances. For each instance, RUL2021 checked only the non-DC status; `FindSRNC` also returned an SRN cycle.

The left-hand plot of Figure 6 shows the average execution times of the two algorithms for each sub-benchmark. These results highlight that computing the SRN cycle does not require significant computational overhead. More interesting is that by analyzing the cycles computed by `FindSRNC`, we can evaluate the characteristics of the non-DC instances in the benchmark. The table in Figure 6 shows that, for each $n$, the average number of edges in the SRN cycle (i.e., the SRN cycle length) is quite small (less than 9); and most instances present a *simple* SRN cycle (i.e., an SRN cycle having no (annotated) bypass edges and, hence, comprising only edges that were already present in the input STNU).

`FindSRNC` outputs a *non-simple* SRN cycle very compactly. However, we also computed the *fully expanded* version of each cycle, recursively replacing each bypass edge by the annotated path from which it was derived. The average length of the expanded cycles increased to a maximum of 16 in each sub-benchmark, revealing that an SRN cycle can involve more edges from the original STNU than one might suspect from the compact version.

Finally, we checked that *no* instance leads to an expanded SRN cycle with *any* repeated edges. Since the benchmark was built to simulate temporal business processes organized on five lanes, the absence of complex SRN cycles in 500 random instances suggests that such instances may only rarely appear in practice; but if they do, `FindSRNC` will find them.

## 5 Conclusion

This paper presented the `FindSRNC` algorithm that modifies the fastest DC-checking algorithm for STNUs to accumulate path information while also rigorously addressing the compact representation of the SRN cycles it outputs. When given an overconstrained STNU, `FindSRNC` can be used to identify constraints to relax or contingent durations to tighten. It can also be used as a supporting process in an iterative algorithm for finding a DC STNU that well approximates a Probabilistic Simple Temporal Network [20, 23, 21, 1].

## References

1 Shyan Akmal, Savanna Ammons, Hemeng Li, and James C. Boerkoel, Jr. Quantifying degrees of controllability in temporal networks with uncertainty. In *29th International Conference on Automated Planning and Scheduling (ICAPS 2019)*, pages 22–30, 2019. URL: `https://ojs.aaai.org/index.php/ICAPS/article/view/3456`, `doi:10.1609/icaps.v29i1.3456`.

2 Nikhil Bhargava, Tiago Vaquero, and Brian C. Williams. Faster conflict generation for dynamic controllability. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17)*, pages 4280–4286, 2017. `doi:10.24963/ijcai.2017/598`.

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

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

5 Johann Eder, Marco Franceschetti, and Josef Lubas. Dynamic Controllability of Processes without Surprises. *Applied Sciences*, 12(3):1461, January 2022. `doi:10.3390/app12031461`.

6 Cheng Fang, Andrew J. Wang, and Brian C. Williams. Chance-constrained Static Schedules for Temporally Probabilistic Plans. *Journal of Artificial Intelligence Research*, 75:1323–1372, 2022. `doi:10.1613/jair.1.13636`.

7 Luke Hunsberger. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *16th International Symposium on Temporal Representation and Reasoning (TIME-2009)*, pages 155–162, 2009. `doi:10.1109/TIME.2009.25`.

8 Luke Hunsberger. Magic Loops in Simple Temporal Networks with Uncertainty–Exploiting Structure to Speed Up Dynamic Controllability Checking. In *5th International Conference on Agents and Artificial Intelligence (ICAART-2013)*, volume 2, pages 157–170, 2013. `doi:10.5220/0004260501570170`.

9 Luke Hunsberger. Magic Loops and the Dynamic Controllability of Simple Temporal Networks with Uncertainty. In Joaquim Filipe and Ana Fred, editors, *Agents and Artificial Intelligence*, volume 449 of *Communications in Computer and Information Science (CCIS)*, pages 332–350, 2014. `doi:10.1007/978-3-662-44440-5_20`.

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

11 Erez Karpas, Steven J. Levine, Peng Yu, and Brian C. Williams. Robust Execution of Plans for Human-Robot Teams. In *25th Int. Conf. on Automated Planning and Scheduling (ICAPS-15)*, volume 25, pages 342–346, 2015. `doi:10.1609/icaps.v25i1.13698`.

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

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

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

**15** Jun Peng, Jingwei Zhu, and Liang Zhang. Generalizing STNU to Model Non-functional Constraints for Business Processes. In *2022 International Conference on Service Science (ICSS)*, pages 104–111. IEEE, May 2022. `doi:10.1109/ICSS55994.2022.00024`.

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

**17** Roberto Posenato. CSTNU Tool: A Java library for checking temporal networks. *SoftwareX*, 17:100905, 2022. `doi:10.1016/j.softx.2021.100905`.

**18** Christoph Strassmair and Nicholas Kenelm Taylor. Human Robot Collaboration in Production Environments. In *23rd IEEE International Symposium on Robot and Human Interactive Communication 2014*, 2014. URL: `https://researchportal.hw.ac.uk/en/publications/human-robot-collaboration-in-production-environments`.

**19** Robert Endre Tarjan. Shortest Paths. Technical report, AT&T Bell Laboratories, 1981.

**20** Ioannis Tsamardinos. A probabilistic approach to robust execution of temporal plans with uncertainty. In *Methods and Applications of Artificial Intelligence (SETN 2002)*, volume 2308 of *Lecture Notes in Artificial Intelligence (LNAI)*, pages 97–108, 2002. `doi:10.1007/3-540-46014-4_10`.

**21** Andrew J. Wang. *Risk-bounded Dynamic Scheduling of Temporal Plans.* PhD thesis, Massachusetts Institute of Technology, 2022. URL: `https://hdl.handle.net/1721.1/147542`.

**22** Peng Yu, Cheng Fang, and Brian Charles Williams. Resolving uncontrollable conditional temporal problems using continuous relaxations. In *24th International Conference on Automated Planning and Scheduling, ICAPS 2014*. AAAI, 2014. URL: `http://www.aaai.org/ocs/index.php/ICAPS/ICAPS14/paper/view/7895`, `doi:10.1609/icaps.v24i1.13623`.

**23** Peng Yu, Brian C. Williams, Cheng Fang, Jing Cui, and Patrick Haslum. Resolving overconstrained temporal problems with uncertainty through conflict-directed relaxation. *Journal of Artificial Intelligence Research*, 60:425–490, 2017. `doi:10.1613/jair.5431`.