New Techniques for Checking Dynamic Controllability of Simple Temporal Networks with Uncertainty

Luke Hunsberger

Computer Science Department, Vassar College
Poughkeepsie, NY, 12604-0444, USA
hunsberg@cs.vassar.edu

Abstract. A Simple Temporal Network with Uncertainty (STNU) is a structure for representing time-points, temporal constraints, and temporal intervals with uncertain—but bounded—durations. The most important property of an STNU is whether it is dynamically controllable (DC)—that is, whether there exists a strategy for executing its time-points such that all constraints will necessarily be satisfied no matter how the uncertain durations turn out. Algorithms for checking from scratch whether STNUs are dynamically controllable are called (full) DC-checking algorithms. Algorithms for checking whether the insertion of one new constraint into an STNU *preserves* its dynamic controllability are called *incremental* DC-checking algorithms. This paper introduces novel techniques for speeding up both full and incremental DC checking. The first technique, called *rotating Dijkstra,* enables constraints generated by propagation to be immediately incorporated into the network. The second uses novel heuristics that exploit the nesting structure of certain paths in STNU graphs to determine good orders in which to propagate constraints. The third technique, which only applies to incremental DC checking, maintains information acquired from previous invocations to reduce redundant computation. The most important contribution of the paper is the incremental algorithm, called *Inky,* that results from using these techniques. Like its fastest known competitors, *Inky* is an $O(N^3)$-time algorithm. However, a comparative empirical evaluation of the top incremental algorithms, all of which have only very recently appeared in the literature, must be left to future work.

1 Introduction

An intelligent agent must be able to plan, schedule and manage the execution of its activities. Invariably, those activities are subject to a variety of temporal constraints, such as release times, deadlines and precedence constraints. In addition, in some domains, the agent may control the starting times for actions, but not their durations [1, 8]. For a simple example, I may control the starting time for my taxi ride to the airport, but not its duration. Although I may know that the ride will last between 15 and 30 minutes, I only *discover* the actual duration in real time, when I arrive at the airport. Therefore, if I need to ensure that I arrive at the airport no later than 10:00, I must start my taxi ride no later than 9:30 to guard against the possibility that the ride might last 30 minutes. In more complicated examples involving large numbers of actions with uncertain durations, generating a succesful *execution strategy* becomes more challenging.

A *Simple Temporal Network with Uncertainty* (STNU) is a data structure that an agent can use to support the planning, scheduling and executing of its activities, some of which may have uncertain durations [11]. The most important property of an STNU is whether it is *dynamically controllable* (DC)—that is, whether there exists a dynamic strategy for executing the constituent actions such that all temporal constraints are guaranteed to be satisfied no matter how the uncertain action durations happen to turn out in real time. The strategy is *dynamic* in that its execution decisions may depend on past execution events, but not on advance knowledge of future events. Algorithms for determining from scratch whether an STNU is DC are called (full) DC-checking algorithms. The fastest DC-checking algorithm reported so far is the $O(N^3)$-time algorithm presented by Morris in 2014 [10], where N is the number of time-points in the network.

In most applications, an STNU is not populated with constraints all at once, but instead one constraint—or a few constraints—at a time. As each new constraint is added to the network, it is important to know whether the dynamic controllability property has been preserved. An *incremental* DC-checking algorithm is an algorithm for determining whether the insertion of a single new (or tighter) constraint into a DC STNU preserves the DC property. Several related incremental DC-checking algorithms have been reported in the literature [17, 16, 13–15]. The latest algorithm in this sequence [15], which is quite similar to Morris' full DC-checking algorithm, also runs in $O(N^3)$ time.

This paper introduces several new techniques for speeding up both full and incremental DC checking. The first technique, called *rotating Dijkstra,* enables constraints generated by propagation to be immediately incorporated into the network. The second uses novel heuristics that exploit the nesting structure of certain paths in STNU graphs to determine good orders in which to propagate constraints. The third, which applies only to incremental DC checking, maintains information from prior invocations to significantly reduce redundant computation.

The full DC-checking algorithm that results from using these techniques is called *Speedy.* A preliminary version of this paper [7] showed that *Speedy* achieves a significant improvement over the $O(N^4)$-time DC-checking algorithm of Morris [9] which, at the time, was the fastest known DC-checking algorithm. Morris subsequently presented his $O(N^3)$-time DC-checking algorithm [10]. It is not known whether *Speedy* will be competitive with this new algorithm, which follows a completely different approach to DC checking. *Speedy* is most likely to be competitive in scenarios involving relatively small numbers of contingent links, which may make it a practical alternative.

The main contribution of this paper is the incremental algorithm, called *Inky,* that results from the new techniques. The worst-case performance of *Inky* is $O(N^3)$, which matches that of the fastest known alternatives [15, 10]. Furthermore, given its reduction of redundant computations, it is expected not only to be competitive, but perhaps much faster than these algorithms. Unfortunately, since the fastest known alternatives either have not yet been published [15] or have only very recently been published [10], a full comparative evaluation of the top competitors must be left to future work.

Fig. 1. The graphs for the STNs discussed in the text

2 Background

This section presents relevant background about Simple Temporal Networks (STNs) and Simple Temporal Networks with Uncertainty (STNUs). The presentation highlights the strong analogies between STNs and STNUs, culminating in the analogous Fundamental Theorems that explicate the relationships between an STN/STNU, its associated graph, and its associated shortest-paths matrix.

2.1 Simple Temporal Networks

A Simple Temporal Network is a pair, $(\mathcal{T},\mathcal{C})$, where $\mathcal{T}$ is a set of real-valued variables called *time-points,* and $\mathcal{C}$ is a set of binary constraints of the form, $Y-X\leq\delta$, where $X,Y\in\mathcal{T}$ and $\delta\in\mathbf{R}$ [3]. An STN is called *consistent* if it has a solution (i.e., a set of values for the time-points that jointly satisfy the constraints). Consider the STN where:

$$\mathcal{T}=\{A,C,X,Y\};\quad \mathcal{C}=\{(C-A\leq 10),(A-C\leq -5),(C-Y\leq 3),(X-C\leq -2)\}.$$

It is consistent. One of its solutions is: $\{(A=0),\ (C=6),\ (X=3),\ (Y=4)\}$.

Each STN, $\mathcal{S}=(\mathcal{T},\mathcal{C})$, has an associated graph, $\mathcal{G}=\langle\mathcal{T},\mathcal{E}\rangle$, where the time-points in $\mathcal{T}$ serve as the nodes for the graph, and the constraints in $\mathcal{C}$ correspond one-to-one to its edges. In particular, each constraint, $Y-X\leq\delta$, in $\mathcal{C}$ corresponds to an edge, $X\xrightarrow{\delta}Y$, in $\mathcal{E}$. The graph for the STN above is shown on the lefthand side of Fig. 1. For convenience, the constraints and edges associated with an STN are called *ordinary* constraints and *ordinary* edges.

Each path in an STN graph, $\mathcal{G}$, corresponds to a constraint that must be satisfied by any solution for the associated STN, $\mathcal{S}$. In particular, if $\mathcal{P}$ is a path from X to Y of length $|\mathcal{P}|$ in $\mathcal{G}$, then the constraint, $Y-X\leq|\mathcal{P}|$, must be satisfied by any solution to $\mathcal{S}$. For example, in the STN from Fig. 1, the path from Y to C to A of length -2 represents the constraint, $A-Y\leq -2$ (i.e., $Y\geq A+2$). The righthand graph in Fig. 1 includes a dashed edge from Y to A that makes this constraint explicit. Note that this *derived* constraint is satisfied by the solution given earlier. Similar remarks apply to the edge from A to X.

Due to these sorts of connections, the all-pairs, shortest-paths (APSP) matrix—called the *distance matrix,* $\mathcal{D}$—plays an important role in the theory of STNs. In fact, the *Fundamental Theorem of STNs* states that the following are equivalent: (1) $\mathcal{S}$ is consistent; (2) each *loop* in $\mathcal{G}$ has non-negative length; and (3) $\mathcal{D}$ has only non-negative entries down its main diagonal [3, 5].

2.2 Simple Temporal Networks with Uncertainty

A Simple Temporal Network with Uncertainty augments an STN to include a set, $\mathcal{L}$, of *contingent links,* each of which represents a temporal interval whose duration is bounded but uncontrollable [11]. Each contingent link has the form, (A,x,y,C), where $A,C \in \mathcal{T}$ and $0 < x < y < \infty$. A is called the *activation* time-point; C is called the *contingent* time-point. Although the link's duration, $C-A$, is uncontrollable, it is guaranteed to lie within the interval, $[x,y]$. When an agent uses an STNU to manage its activities, contingent links typically represent actions with uncertain durations. The agent may control the action's starting time (i.e., when A executes), but only *observes,* in real time, the action's ending time (i.e., when C executes).[1] For example, consider the STNU:

$$\mathcal{T} = \{A,C,X,Y\}; \quad \mathcal{C} = \{(C-Y \leq 3),\ (X-C \leq -2)\}; \quad \mathcal{L} = \{(A,5,10,C)\}.$$

It is similar to the STN seen earlier, except for one important difference. In the STN, the duration, $C-A$, was constrained to lie within the interval $[5,10]$, but the agent was free to choose any values for A and C that satisfied that constraint. In contrast, in the STNU, $C-A$ is the duration of a contingent link. This duration is guaranteed to lie within $[5,10]$, but the agent does not get to choose this value. For example, if A is executed at 0, then the agent only gets to *observe* the execution of C when it happens, sometime between 5 and 10. In this sense, the contingent duration is uncontrollable, but bounded.

Dynamic Controllability. For an STNU, $(\mathcal{T},\mathcal{C},\mathcal{L})$, the most important property is whether it is *dynamically controllable* (DC)—that is, whether there exists a *strategy* for executing the controllable (i.e., non-contingent) time-points in $\mathcal{T}$ such that all constraints in $\mathcal{C}$ are guaranteed to be satisfied no matter how the durations of the contingent links in $\mathcal{L}$ turn out in real time—within their specified bounds [11]. Such strategies, if they exist, are called *dynamic execution strategies—dynamic* in that their execution decisions may depend on the observation of past execution events, but not on advance knowledge of future events. It is not hard to verify that the following strategy is an example of a dynamic execution strategy for the sample STNU:

- Execute A at 0, and X at 3.
- If C executes *before* time 7, then execute Y at time $C+1$; otherwise, execute Y at 7.

Thus, the sample STNU is dynamically controllable. The given strategy is dynamic in that the decision to execute Y depends on observations about C.

STNU graphs. Each STNU, $(\mathcal{T},\mathcal{C},\mathcal{L})$, has an associated graph, $\langle \mathcal{T},\mathcal{E}^+ \rangle$, 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 give rise to the edges in $\mathcal{E}^+$ [12]. To capture the difference between constraints and contingent links, the edges in $\mathcal{E}^+$ come in two varieties: ordinary and labeled. As with an STN, each constraint, $Y-X \leq \delta$, in $\mathcal{C}$ corresponds to an ordinary edge, $X \xrightarrow{\delta} Y$, in $\mathcal{E}^+$. In addition, each contingent link, (A,x,y,C), in $\mathcal{L}$ gives rise to two ordinary edges that together represent the constraint, $C-A \in [x,y]$. Finally, each contingent link, (A,x,y,C), also gives rise to the following *labeled* edges:

[1] Agents are not part of the semantics of STNUs. They are used here for expository convenience.

Fig. 2. The graph for the sample STNU before (left) and after (right) generating new edges

(No Case)	$A \xleftarrow{x} C \xleftarrow{y} D$	adds:	$A \xleftarrow{x+y} D$
(Lower Case)	$A \xleftarrow{x} C \xleftarrow{c:y} D$	adds:	$A \xleftarrow{x+y} D$
(Upper Case)	$A \xleftarrow{B:x} C \xleftarrow{y} D$	adds:	$A \xleftarrow{B:x+y} D$
(Cross Case)	$A \xleftarrow{B:x} C \xleftarrow{c:y} D$	adds:	$A \xleftarrow{B:x+y} D$
(Label Removal)	$B \xleftarrow{b:x} A \xleftarrow{B:z} C$	adds:	$A \xleftarrow{z} C$

Table 1. Morris and Muscettola's edge-generation rules for STNUs

a *lower-case* edge, $A \xrightarrow{c:x} C$, and an *upper-case* edge, $A \xleftarrow{C:-y} C$. The lower-case (LC) edge represents the *uncontrollable possibility* that the duration, $C-A$, might assume its lower bound, x. The upper-case (UC) edge represents the *uncontrollable possibility* that $C-A$ might assume its upper-bound, y. The graph for the sample STNU is shown on the lefthand side of Fig. 2.

Edge generation for STNUs. Because the labeled edges in an STNU graph represent uncontrollable possibilities, edge generation (equiv., constraint propagation) for STNUs is more complex than for STNs. In particular, a variety of rules are required to handle the interactions between different kinds of edges. Table 1 lists the edge-generation rules for STNUs given by Morris and Muscettola (2005).[2] The *No Case* rule encodes ordinary STN constraint propagation. The *Lower Case* rule generates edges/constraints that guard against the possibility of a contingent link taking on its minimum duration. The *Upper Case* rule generates edges/constraints that guard against the possibility of a contingent link taking on its maximum duration. The *Cross Case* rule addresses the interaction of LC and UC edges from different contingent links.

The edge-generation rules are *sound* in the sense that the edges they generate correspond to constraints that must be satisfied by any dynamic execution strategy. For example, consider the righthand graph in Fig. 2. For ease of exposition, suppose that A has been executed at 0. The edge, $C \xrightarrow{-2} X$, represents the constraint, $X - C \leq -2$ (i.e., $X \leq C - 2$), which requires X to be executed *before* the contingent time-point C.

[2] The rules are shown using Morris and Muscettola's notation. Note that: the x's and y's here are not necessarily bounds for contingent links; C is only required to be contingent in the *Lower Case* and *Cross Case* rules, where its activation time-point is D and its *lower* bound is y; and in the *Upper Case* and *Cross Case* rules, B is contingent, with activation time-point A. The *Lower Case* rule only applies when $x \leq 0$ and $A \neq C$; the *Cross Case* rule only applies when $x \leq 0$ and $B \neq C$; and the *Label Removal* rule only applies when $z \geq -x$.

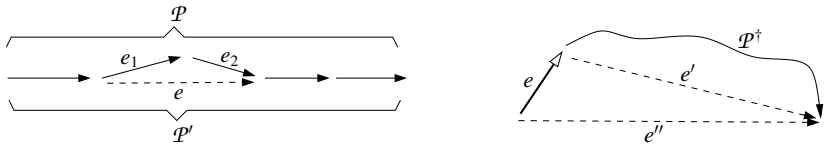

Fig. 3. (a) A generic path transformation; (b) reducing away a lower-case edge

To ensure that this constraint will be satisfied even if C eventually happens to take on its minimum value of 5, X must be executed no later than 3 units after A, whence the dashed edge from A to X. This dashed edge can be generated by applying the *Lower Case* rule to the path from A to C to X.

Next, consider the edge, $Y \xrightarrow{3} C$, which represents the constraint, $C - Y \leq 3$ (i.e., $Y \geq C - 3$). To ensure that this constraint is satisfied, the following *conditional* constraint must be satisfied: *While C remains unexecuted, Y must occur* 7 *or more units after A*. This conditional constraint, represented by the upper-case edge, $Y \xrightarrow{C:-7} A$, effectively guards against the possibility of C taking on its *maximum* value, 10. The UC edge can be generated by applying the *Upper Case* rule to the path from Y to C to A.

It is not hard to verify that the constraints corresponding to these generated edges are satisfied by the sample dynamic execution strategy given earlier.

Semi-reducible paths. Recall that, for an STN, each path in its graph corresponds to a constraint that must be satisfied by any solution to that STN. In contrast, for an STNU, it is the *semi-reducible* paths—defined below—that correspond to the (possibly conditional) constraints that must be satisfied by any dynamic execution strategy for that STNU. Whereas an STN is consistent if and only if its graph has no negative-length loops, an STNU is dynamically controllable if and only if its graph has no *semi-reducible* negative-length loops [9].

Before defining semi-reducible paths, it is useful to view the edge-generation rules from Table 1 as *path-transformation* rules, as follows. Suppose e_1 and e_2 are consecutive edges in a path $\mathcal{P}$, and that one of the first four rules can be applied to e_1 and e_2 to generate a new edge e, as illustrated in Fig. 3a, where $\mathcal{P}'$ is the path obtained from $\mathcal{P}$ by replacing the edges, e_1 and e_2, with e. We say that $\mathcal{P}$ has been transformed into $\mathcal{P}'$. Similar remarks apply to the *Label Removal* rule, which operates on a single edge.

A path in an STNU graph is called *semi-reducible* if it can be transformed into a path that has only OU-edges [9]. Thus, for any semi-reducible path $\mathcal{P}$, there must be some transformation of $\mathcal{P}$ whereby each lower-case edge e in $\mathcal{P}$ is eventually "reduced away" by either the *Lower Case* or *Cross Case* rule. In other words, as illustrated in Fig. 3b, for each lower-case edge e in $\mathcal{P}$, there must be some sub-path, $\mathcal{P}^\dagger$, following e in $\mathcal{P}$, such that $\mathcal{P}^\dagger$ can be transformed into a single edge, e', using the rules from Table 1. Depending on whether e' is ordinary or upper-case, either the *Lower Case* or *Cross Case* rule can then be used to transform e and e' into a single edge, e'', effectively reducing away the lower-case edge e.

The soundness of the edge-generation rules ensures that the constraints represented by semi-reducible paths must be satisfied by any dynamic execution strategy. Since

Fig. 4. The OU-graphs, $\mathcal{G}^{\text{ou}}$, for the corresponding graphs from Fig. 2

Fig. 5. The *AllMax* graphs, $\mathcal{G}_x$, for the corresponding graphs from Fig. 2

shorter paths correspond to stronger constraints, the all-pairs, shortest-*semi-reducible*-paths (APSSRP) matrix, $\mathcal{D}^*$, plays an important role in the theory of STNUs. The *Fundamental Theorem of STNUs* states that the following are equivalent for any STNU $\mathcal{S}$: (1) $\mathcal{S}$ is dynamically controllable; (2) every *semi-reducible* loop in its associated graph has non-negative length; and (3) its APSSRP matrix, $\mathcal{D}^*$, has only non-negative entries along its main diagonal [9, 6].

2.3 Morris' $O(N^4)$-time DC-Checking Algorithm

In 2006, Morris presented an $O(N^4)$-time DC-checking algorithm—hereinafter called the *Morris-*N^4 algorithm—that uses the rules from Table 1 to generate new edges. Each newly generated edge is added not only to $\mathcal{G}$, but also, in a stripped down form, to a related *STN* graph, called the *AllMax* graph. If the *AllMax* graph ever exhibits a negative-length loop, the original STNU is declared to be non-DC. The algorithm achieves its efficiency by focusing its edge-generation activity on reducing away lower-case edges. Although Morris has since presented a faster, $O(N^3)$-time algorithm [10], the Morris-N^4 algorithm lays a foundation for the rest of this paper and, so, is summarized below.

Let $\mathcal{S}$ be an STNU and $\mathcal{G}$ its associated graph. The lengths of all shortest *semi-reducible* paths in $\mathcal{G}$ can be determined as follows. First, for convenience, any *ordinary* or *upper-case* edge may be called an *OU-edge,* and the graph consisting of all of the OU-edges from $\mathcal{G}$ shall be called the *OU-graph* for $\mathcal{G}$, denoted by $\mathcal{G}^{\text{ou}}$. Since the edges in $\mathcal{G}^{\text{ou}}$ are drawn from $\mathcal{G}$, any path in $\mathcal{G}^{\text{ou}}$ also appears in $\mathcal{G}$. In addition, since each path in $\mathcal{G}^{\text{ou}}$ contains only OU-edges, it is trivially semi-reducible. Thus, the paths in $\mathcal{G}^{\text{ou}}$ are a subset of the semi-reducible paths in $\mathcal{G}$. Furthermore, since the rules from Table 1 generate only OU-edges, inserting any such edges into *both* $\mathcal{G}^{\text{ou}}$ and $\mathcal{G}$ necessarily preserves the property that the paths in $\mathcal{G}^{\text{ou}}$ are a subset of the semi-reducible paths in $\mathcal{G}$. For example, Fig. 4 shows the OU-graph for the sample STNU from Fig. 2 before (left) and after (right) the insertion of two newly generated edges.

Next, with the goal of computing the *lengths* of the paths in $\mathcal{G}^{\text{ou}}$, let $\mathcal{G}_x$ be the graph obtained by removing the *alphabetic labels* from all upper-case edges in $\mathcal{G}^{\text{ou}}$.

$\mathcal{G}_x$ is called the *AllMax* graph because it can be obtained from the original STNU by forcing each contingent link to take on its maximum value [12]. The *AllMax* graphs corresponding to the graphs from Fig. 2 are shown in Fig. 5. In the figure, the ordinary edge, $C \xrightarrow{-5} A$, is drawn in light gray because it represents a weaker constraint than the edge, $C \xrightarrow{-10} A$, and thus can be ignored. Note that if the edge-generation rules produce an upper-case edge (e.g., the UC edge from Y to A in Fig. 4), then that edge is stripped of its alphabetic label before being added to the *AllMax* graph, $\mathcal{G}_x$.

Since the *AllMax* graph contains only ordinary edges, it is an *STN* graph. Its associated distance matrix, $\mathcal{D}_x$, is called the *AllMax* matrix. For any X and Y, $\mathcal{D}_x(X,Y)$ equals the length of a shortest path from X to Y in $\mathcal{G}_x$. $\mathcal{D}_x(X,Y)$ also equals the length of a shortest *semi-reducible* path from X to Y in the OU-graph, $\mathcal{G}^{\text{ou}}$. Because $\mathcal{G}^{\text{ou}}$ may contain only a subset of the semi-reducible paths from $\mathcal{G}$, $\mathcal{D}_x(X,Y)$ only provides an *upper bound* on the length of the shortest semi-reducible path from X to Y in $\mathcal{G}$. However, as newly generated edges are inserted into the appropriate graphs, the upper bounds on the lengths of shortest semi-reducible paths provided by $\mathcal{D}_x$ typically tighten.

The Morris-N^4 algorithm focuses its attention on finding paths in the graph that can be used to reduce away lower-case edges. However, it need not find all such paths; instead, it suffices to follow shortest *allowable* paths (defined below), seeking to find *extension sub-paths* (defined below). The following sequence of definitions is provided for expository convenience. First, let e be any lower-case edge, $A \xrightarrow{c:x} C$. A *quasi-allowable* path for e is any path, $\mathcal{P}_e$, such that:

- $\mathcal{P}_e$ is a loopless path emanating from C;
- $\mathcal{P}_e$ contains only OU-edges (i.e., edges in $\mathcal{G}^{\text{ou}}$); and
- $\mathcal{P}_e$ is *breach-free* (i.e., does *not* contain any upper-case edges labeled by C).[3]

Next, the path $\mathcal{P}_e$ is said to have the *positive proper prefix* (PPP) property if every *proper* prefix of $\mathcal{P}_e$ has *positive* length. A quasi-allowable path that has the PPP property is called an *allowable* path. If, further still, $\mathcal{P}_e$ itself has non-positive length, then $\mathcal{P}_e$ is called an *extension sub-path* for e. It is not hard to show that any extension sub-path for e can be transformed into a single edge, e', which can then be used to reduce away e, as described earlier (cf. Fig. 3b). The resulting edge, e'', is then inserted into $\mathcal{G}^{\text{ou}}$ and—after removing any alphabetic label—$\mathcal{G}_x$. Furthermore, it is not necessary to search through paths that are quasi-allowable but not allowable, since any such path must have a proper prefix that is an extension sub-path. Of course, reducing away a lower-case edge e_1 might generate a new edge E_1 that could subsequently be used as part of an extension sub-path that reduces away another lower-case edge e_2, to generate another new edge, E_2. In such a case, the extension sub-path that generated E_1 is said to be *nested* inside the extension sub-path that generated E_2. However, Morris proved that, for an STNU with K contingent links, it suffices to consider at most K levels of such nesting. As a result, the Morris-N^4 algorithm performs at most K rounds of searching through *shortest* allowable paths to determine whether the original graph, $\mathcal{G}$, contains any semi-reducible negative loops. Pseudo-code for the Morris-N^4 DC-checking algorithm is given in Table 2. Its most important features are:

[3] A breach edge could prevent application of the *Cross Case* rule.

Input: `G`, a graph for an STNU with K contingent links.
Output: `True` if the corresponding STNU is dynamically controllable; `False` otherwise.

```
-1. Gou := OU-graph for G.
 0. Gx  := AllMax graph for G.
 1. for i = 1, K:                                          (Outer Loop)
 2.   result := Bellman_Ford_SSSP(Gx).
 3.   if (result == inconsistent) return False.
 4.   else generate_potential_function(result).
 5.   newEdges := {}.
 6.   for j = 1, K:                                        (Inner Loop)
 7.     Let C_j be the j^th contingent time-point.
 8.     Traverse shortest allowable paths in G^ou emanating from C_j, searching for
        extension sub-paths that generate new edges. Add new edges to newEdges.
 9.   end for j = 1, K.
10.   if newEdges empty, return True.
11.   else insert newEdges into Gou and Gx.
12. end for i = 1, K.
13. result := Bellman_Ford_SSSP(Gx).
14. if (result == inconsistent) return False.
15. else return True.
```

Table 2. Pseudo-code for the Morris-N^4 DC-checking algorithm

- The outer loop (Lines 1–12) runs at most K times.
- Each outer iteration begins (Line 2) by applying the Bellman-Ford single-source, shortest-paths (SSSP) algorithm [2] to the *AllMax* graph $\mathcal{G}_x$. This serves two purposes. First, if Bellman-Ford determines that the *AllMax* graph is inconsistent, then the algorithm immediately returns `False`. However if $\mathcal{G}_x$ *is* consistent, then the shortest-path information generated by Bellman-Ford can be used to create a *potential function* (Line 4) to transform the lengths of all edges in $\mathcal{G}_x$—and hence all edges in $\mathcal{G}^{\text{ou}}$—to non-negative values, as in Johnson's algorithm [2].
- During each iteration of the outer loop, the inner loop (Lines 6–9) runs exactly K times, once per contingent link.
- The j^{th} iteration of the inner loop (Lines 7–8) focuses on C_j, the contingent time-point for the j^{th} contingent link. The algorithm uses the potential function generated in Line 4 to enable a Dijkstra-like traversal of shortest allowable paths emanating from C_j in the graph $\mathcal{G}^{\text{ou}}$.
- New edges generated by K iterations of the *inner* loop are accumulated in a set, `newEdges` (Line 8). Afterward, if it is discovered that no new edges have been generated, then the algorithm immediately returns `True` (Line 10). On the other hand, if some new edges were generated by the inner loop, then they are inserted into the graphs (Line 11) in preparation for the running of Bellman-Ford at the beginning of the next iteration of the outer loop (Line 2).

- If, after completing K iterations of the *outer* loop, the *AllMax* graph remains consistent (Lines 14–15), then the network must be DC.[4]

The complexity of the algorithm is dominated by the $O(N^3)$-time complexity of the Bellman-Ford algorithm (Line 2), as well as the Dijkstra-like traversals of shortest allowable paths (Line 8). Since Bellman-Ford is run a maximum of K times, and $O(K) = O(N)$, the overall complexity due to the use of Bellman-Ford is $O(N^4)$. Each Dijkstra-like traversal of shortest allowable paths (Line 8) is $O(N^2)$ in the worst case. Since these traversals are run at most K^2 times, the overall contribution is again $O(N^4)$.

3 *Speedy*: Speeding Up Full DC Checking

As discussed above, the Morris-N^4 algorithm uses the Bellman-Ford algorithm to compute a potential function at the beginning of each iteration of the outer loop (Lines 2–4). This same potential function is then used for all K iterations of the inner loop (Lines 6–9). For this reason, any new edges discovered during the K iterations of the inner loop cannot be inserted into $\mathcal{G}^{\text{ou}}$ or $\mathcal{G}_x$ until preparing for the next iteration of the *outer* loop (Line 11). To see this, consider that the Dijkstra-like traversal of shortest allowable paths (Line 8) depends on all edge-lengths having been converted into non-negative values by the potential function. Incorporating new edges into this traversal without recomputing the potential function could introduce negative-length edges, violating the conditions of a Dijkstra-like traversal. A second important consequence of delaying the integration of new edges until the next *outer* iteration, is that the *order* in which the contingent links are processed by the inner loop cannot make any difference to the Morris-N^4 algorithm.

Given these observations, Hunsberger [7] made two inter-related modifications to the Morris-N^4 algorithm to significantly improve its performance. For convenience, the modified algorithm will hereinafter be called *Speedy.* The first modification used by *Speedy* is called *rotating Dijkstra.* It enables the edges generated by one iteration of the *inner* loop to be *immediately* inserted into the network for use during the very next iteration of the *inner* loop, instead of waiting until the beginning of the next *outer* iteration. Next, because each iteration of the inner loop can use the edges generated by any prior iteration, *Speedy* uses a heuristic function to choose a "good" order in which to process the lower-case edges during the K iterations of the inner loop. The heuristic is inspired by the nesting structure of so-called *magic loops* analyzed in prior work [6]. In some networks, these two changes combine to produce an order-of-magnitude speed-up in DC checking [7]. Although Morris has, in the interim, presented an $O(N^3)$-time DC-checking algorithm that *might* be faster than *Speedy,* the techniques used by *Speedy* also have novel applications to *incremental* DC checking, to be discussed in Section 4; therefore, these new techniques are briefly summarized below.

Recalling Johnson's algorithm. Johnson's algorithm [2] is an all-pairs, shortest-paths algorithm that can be used on graphs whose edges have any numerical lengths: positive, negative or zero. It begins by using the Bellman-Ford single-*source*, shortest-paths

[4] This conclusion is justified by Morris' Theorem 3 that an STNU contains a semi-reducible negative loop if and only if it contains a *breach-free semi-reducible negative loop* in which the extension sub-paths are nested to a depth of at most K [9].

algorithm to generate a potential function, h. In particular, for any node X, $h(X)$ is defined to be the length of the shortest path from some source node S to X. Johnson's algorithm then uses that potential function to convert edge lengths to non-negative values, as follows. For any edge, $U \xrightarrow{\delta} V$, the converted length is $h(U)+\delta-h(V)$. This is guaranteed to be non-negative since the path from S to V via U cannot be shorter than the shortest path from S to V. Then, for each time-point X in the graph, Johnson's algorithm runs Dijkstra's single-*source*, shortest-paths algorithm on the re-weighted edges using X as the source. This works because shortest paths in the re-weighted graph correspond to shortest paths in the original graph. In particular, for any X and Y, the length of the shortest path from X to Y in the original graph is $h(Y)+D(X,Y)-h(X)$, where $D(X,Y)$ is the length of the shortest path from X to Y in the re-weighted graph.

Rotating Dijkstra. The rotating Dijkstra technique is based on several observations. First, just as single-*source,* shortest-paths information can be used to generate a potential function to support the conversion of edge-lengths to non-negative values, so too can single-*sink,* shortest-paths information be used in this way [5]. Furthermore, whether the re-weighting of edges is done using a source-based or sink-based potential function, Dijkstra's algorithm can be run on the re-weighted graph following edges forward from a single source or following edges backward from a single sink. This paper refers to the different combinations as *sinkPot/srcDijk, sinkPot/sinkDijk,* and so on.

Second, when a contingent link, (A,x,y,C), is being processed during one iteration of the inner loop of the Morris-N^4 algorithm, any new edge generated during that iteration must have that link's activation time-point, A, as its source.[5] However, adding edges whose *source* time-point is A cannot cause changes to the lengths of shortest paths *terminating* in A [5]. Thus, adding new edges whose source is A cannot cause any changes to entries of the form, $\mathcal{D}_x(T,A)$, for any time-point T. As a result, if the potential function used to re-weight the edges for this Dijkstra-like traversal is a sink-based potential function with A as its sink, then adding new edges generated by that traversal cannot cause any changes to that potential function. Thus, that same potential function can be used along with Dijkstra's single-*sink,* shortest-paths algorithm to re-compute the values, $\mathcal{D}_x(T,A')$, for any time-point T, in preparation for the next iteration of the inner loop, where A' is the activation time-point for the next contingent link to be processed.

Given these observations, the rotating Dijkstra technique takes the following steps to support the Dijkstra-like traversal of shortest allowable paths emanating from the contingent time-point C associated with the contingent link (A,x,y,C).

(1) Given: All entries, $\mathcal{D}_x(T,A)$, for all time-points T. This collection of entries provides a *sink*-based potential function, h_A, where A is the sink.
(2) Use h_A to convert all edge-lengths in $\mathcal{G}^{\text{ou}}$ to non-negative values in preparation for a *source*-Dijkstra traversal of shortest allowable paths emanating from C, as in the Morris-N^4 algorithm (Line 8).

[5] This follows immediately from how new edges are generated [9]. In particular, each new edge is generated by reducing the path consisting of the lower-case edge, $A \xrightarrow{c:x} C$, and some extension sub-path into a single new edge. Since such a reduction preserves the endpoints of the path, the generated edge must have A as its source.

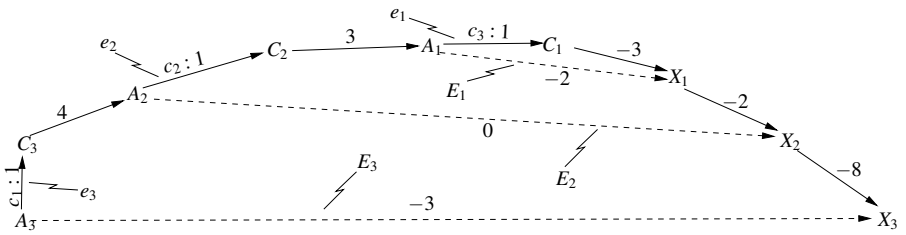

Fig. 6. A path with nested extension sub-paths

(3) Since any edges generated by this traversal must have A as their source, the new edges cannot cause any changes to the sink-based potential function, h_A. Thus, h_A can subsequently be used to support a *sinkDijkstra* computation of all entries of the form $\mathcal{D}_x(T,A')$, for any T, where A' is the activation time-point for the next contingent link to be processed. This computation is abbreviated as *sinkPot/sinkDijk*(A,A'), since A is the sink for the potential function, and A' is the sink for the Dijkstra traversal. It generates the potential function, $h_{A'}$, for the next iteration.

For the very first iteration of the inner loop, the entries, $\mathcal{D}_x(T,A)$, needed in Step 1 are provided by an initial run of Johnson's algorithm. For every subsequent iteration, the information needed in Step 1 is obtained from Step 3 of the preceding iteration.

Choosing an order in which to process the contingent links. Because the rotating Dijkstra technique enables newly generated edges to be inserted into the network immediately, rather than waiting for the next iteration of the *outer* loop, edges generated by one iteration of the inner loop can be used by the very next iteration. Thus, subsequent iterations of the inner loop may generate new edges sooner than they would in the Morris-N^4 algorithm, which can significantly improve performance.

Consider the path shown in Fig. 6. The innermost sub-path, from A_1 to X_1, reduces to (i.e., can be transformed into) a new edge, E_1, from A_1 to X_1. In turn, that enables the next innermost sub-path, from A_2 to X_2, to be reduced to a new edge, E_2, from A_2 to X_2. Finally, that then enables the outermost path, from A_3 to X_3, to be reduced to a single new edge, E_3, from A_3 to X_3. Thus, in this example, if the contingent links are processed in the order, C_1, C_2, C_3, then all three edges, E_1, E_2 and E_3, will be generated in *one* iteration of the outer loop—involving *three* iterations of the inner loop. However, if the contingent links are processed in the opposite order, then *three* iterations of the outer loop—involving *nine* iterations of the inner loop—will be required to generate E_1, E_2 and E_3. To see this, notice that if C_3 is processed first, then the edges E_1 and E_2 will not have been generated yet. And, since allowable paths do not include lower-case edges, the initial search through allowable paths emanating from C_3 will not yield any new edges. Similarly, the initial search through allowable paths emanating from C_2 will not yield any new edges. Only the processing of C_1 will yield a new edge—namely, E_1—during the first iteration of the outer loop. During the second iteration of the outer loop, the processing of C_2 will yield the edge E_2. Finally, during the third iteration of the outer loop, the processing of C_3 will yield the edge E_3. Crucially, since the Morris-

N^4 algorithm does not insert new edges until the beginning of the next iteration of the outer loop, that algorithm would exhibit the same behavior for this path. In general, the Morris-N^4 algorithm requires d iterations of the outer loop to generate new edges arising from semi-reducible paths in which extension sub-paths are nested to a depth d.

Nesting order. Prior work [6] has defined a *nesting order* for semi-reducible paths as follows. Suppose $e_1, e_2, \ldots, e_n$ are the lower-case edges that appear in a semi-reducible path $\mathcal{P}$. Then that ordering of those edges constitutes a *nesting order* for $\mathcal{P}$ if $i < j$ implies that no extension sub-path for e_j is nested within an extension sub-path for e_i in $\mathcal{P}$. For example, the path shown in Fig. 6 has a nesting order e_1, e_2, e_3. The relevance of a nesting order to DC checking is that if a semi-reducible path $\mathcal{P}$ has extension sub-paths nested to a depth d, with a nesting order, $e_1, e_2, \ldots, e_d$, and the lower-case edges (i.e., the contingent links) are processed in that order using the rotating Dijkstra technique, then only *one* iteration of the outer loop will be necessary to generate all edges derivable from $\mathcal{P}$. For the purposes of this paper, it is not necessary to prove this result—although it follows quite easily from the definitions involved—because it is not claimed that for any STNU graph there is a single nesting order that applies to all semi-reducible paths in that graph. However, it does suggest that it might be worthwhile to spend some modest computational effort to find a "good" order in which to process the contingent links in the inner loop of the algorithm.

Toward that end, suppose that e_1 and e_2 are lower-case edges corresponding to the contingent links, (A_1, x_1, y_1, C_1) and (A_2, x_2, y_2, C_2). Suppose further that $\mathcal{P}$ is a shortest allowable path emanating from e_2; and that $\mathcal{P}$ contains e_1. That is, e_1 is nested inside e_2 (e.g., as illustrated in Fig. 6). Although allowable paths for e_2 cannot include any upper-case edges labeled by C_2, the OU-graph invariably includes at least one upper-case edge labeled by C_2. Thus, the *AllMax* matrix entry, $\mathcal{D}_x(C_2, A_1)$, is not a perfect substitute for the length of the shortest *allowable* path from C_2 to A_1. Instead, $\mathcal{D}_x(C_2, A_1)$ is a *lower* bound on that length. Nonetheless, *Speedy*'s heuristic uses it as an imperfect substitute.

The heuristic, H. Let $\mathcal{G}$ be the graph for an STNU with K contingent links, and $\mathcal{G}_x$ the corresponding *AllMax* graph. Run Johnson's algorithm on $\mathcal{G}_x$ to generate the *AllMax* matrix $\mathcal{D}_x$. For each i, let $Q(i)$ be the number of entries of the form $\mathcal{D}_x(C_i, A_j)$ that are non-positive. Let $H(\mathcal{G})$ be a permutation, $\sigma_1, \sigma_2, \ldots, \sigma_K$, such that $r < s$ implies $Q(\sigma_r) \leq Q(\sigma_s)$. In other words, $H(\mathcal{G})$ is obtained by sorting the numbers $1, 2, \ldots, K$ according to the corresponding Q values.

The Speedy *Algorithm.* Pseudo-code for the *Speedy* DC-checking algorithm is given in Table 3. The algorithm first constructs the OU-graph, $\mathcal{G}^{\text{ou}}$, and the *AllMax* graph, $\mathcal{G}_x$ (Lines -1 and 0). It then uses Johnson's algorithm to compute the *AllMax* matrix, $\mathcal{D}_x$ (Line 1). As discussed above, $\mathcal{D}_x$ is used during the computation of the heuristic function, H (Line 2), which determines the processing order for the contingent links. These $\mathcal{D}_x$ entries also provide the potential function in Line 9 during the very first iteration of the inner loop. `GlobalIters`, initially 0 (Line 3), counts the total number of iterations of the inner loop. If this counter ever reaches K^2, the algorithm terminates, returning

Input: `G`, a graph for an STNU with K contingent links.
Output: `True` if the corresponding STNU is dynamically controllable; `False` otherwise.

-1. `Gou := OU-graph for G;`
0. `Gx  := AllMax graph for G;`
1. `Dx  := Johnson(Gx);`
2. `Order := Heuristic(Dx);`
3. `GlobalIters := 0;`
4. `LocalIters := 0;`
5. `for i = 1, K:` (Outer Loop)
6. `  for j = 1, K:` (Inner Loop)
7. `    newEdges := {};`
8. Let (A_j, x_j, y_j, C_j) be the j^{th} contingent link according to `Order`.
9. Use the values, $\mathcal{D}_x(T, A_j)$, as a sink-based potential function, h_{A_j}, to transform the edge-lengths in `Gx` and `Gou` to non-negative values.
10. Traverse shortest *allowable* paths in $\mathcal{G}^{\text{ou}}$ emanating from C_j, searching for *extension sub-paths* that generate new edges. Add such edges to `newEdges`.
11. `for each edge` $A_j \xrightarrow{\delta} X$ `in newEdges: if` $(\delta < -\mathcal{D}_x(X, A_j))$ `return False;`
12. `    GlobalIters++;`
13. `    if (GlobalIters` $\geq K^2$`) return True;`
14. `    elseif newEdges empty:`
15. `      LocalIters++;`
16. `      if (LocalIters` $\geq K$`) return True;`
17. `    else LocalIters := 0;`
18. `    run sinkPotSinkDijk(Aj,A'),` where A' is act'n. time-point for next contingent link.
19. `  end for j = 1, K.`
20. `end for i = 1, K.`
21. `return True.`

Table 3. Pseudo-code for the *Speedy* DC-checking algorithm

`True` (Line 13).[6] `LocalIters`, initially 0 (Line 4), counts the number of *consecutive* iterations of the inner loop since the last time a new edge was generated. If this counter ever reaches K, then the algorithm terminates, returning `True` (Line 16).[7]

In the *Speedy* algorithm, the inner loop (Lines 6–19) cycles through the contingent links in the order given by the heuristic until a termination condition is reached. Since potential functions are re-computed after each *inner* iteration (Line 18), the outer loop is provided only for counting purposes. One iteration of the inner loop spans Lines 7–18. (A_j, x_j, y_j, C_j) is the contingent link to be processed, as determined by `Order`. For the very first iteration of the inner loop, the values, $\mathcal{D}_x(X, A_j)$, that constitute a sink-based potential function (Line 9), are provided by Johnson's algorithm (Line 1); for every other iteration of the inner loop, these values are provided by the *sinkPot/sinkDijk*

[6] This termination condition is analogous to the Morris-N^4 algorithm terminating after K iterations of the *outer* loop.

[7] This termination condition is analogous to the Morris-N^4 algorithm terminating whenever any iteration of the outer loop fails to generate a new edge.

computation from Line 18 of the previous iteration. This potential function is then used in Line 10 to support a srcDijk traversal of shortest allowable paths emanating from C_j. Because the values, $\mathcal{D}_x(X,A_j)$, are available for all time-points X, any new edge from A_j to some X can be immediately checked for consistency (Line 11). If all new edges are judged to be consistent, then the algorithm increments the global counter and checks the two termination conditions (Lines 13 and 16). If no termination condition is reached yet, then a sinkPot/sinkDijk computation is run (Line 18), to compute all entries of the form, $\mathcal{D}_x(X,A')$, where A' is the activation time-point for the contingent link to be processed during the next iteration of the inner loop. These values will form the potential function in Line 9 during the next iteration.

The *Speedy* algorithm was evaluated empirically against the Morris-N^4 algorithm [7], which was, at the time, the fastest DC-checking algorithm in the literature. In the special case of so-called *magic loops,* which represent one kind of worst-case scenario involving maximum nesting of lower-case edges [5], the *Speedy* algorithm exhibited an order-of-magnitude improvement over the Morris-N^4 algorithm, which suggests it would be competitive with the more recent $O(N^3)$-time algorithm. For other networks, the degree of improvement appeared to be proportional to the degree of maximal nesting, indicating that the heuristic was working effectively. Furthermore, using the *reverse* of the order suggested by the heuristic lead to significantly worse performance, again indicating that the heuristic was working effectively.

4 *Inky:* Speeding up Incremental DC Checking

This section presents a new incremental DC-checking algorithm, called *Inky,* that applies similar kinds of insights and techniques seen in the previous section to the problem of incremental DC checking. Toward that end, let $\mathcal{S}$ be an STNU that is known to be dynamically controllable; let $\mathcal{G}$ be its graph; and let $X \xrightarrow{\delta} Y$ be a new edge. The incremental DC-checking problem is to determine whether inserting the new edge into the network will preserve its dynamic controllability.

To avoid redundant computations across multiple invocations, and to bound the number of rounds of edge generation, the *Inky* algorithm maintains an auxiliary K-by-N matrix, called $\mathcal{D}^+$, where for each contingent time-point C and each time-point T, $\mathcal{D}^+(C,T)$ equals the length of the shortest *quasi-allowable* path from C to T (cf. Section 2.3). Given a new edge from X to Y, the *Inky* algorithm begins by sorting the lower-case edges such that the values of $\mathcal{D}^+(C_i,X)$ are *non-increasing.* The reason, as shown by the following theorem, is that processing the lower-case edges in this order will require only *one* pass of the outer loop of the DC-checking algorithm, thereby ensuring that the algorithm runs in $O(N^3)$ time.

Theorem 1. *Let $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ be a dynamically controllable STNU having K contingent links; and let $\mathcal{G}$ be the graph for $\mathcal{S}$. Let E be a new edge of length δ from X to Y, where $X,Y \in \mathcal{T}$. Let $\mathcal{G}^\dagger$ be the graph obtained by inserting the new edge E into $\mathcal{G}$. As illustrated in Fig. 7, let $\mathcal{P}$ be any semi-reducible path in $\mathcal{G}^\dagger$ whose first edge is a lower-case edge e_i: $A_i \xrightarrow{c_i:x_i} C_i$; let U be the final time-point in $\mathcal{P}$; let $\mathcal{P}_i$ be the extension sub-path for e_i in $\mathcal{P}$; and, under the supposition that $\mathcal{P}_i$ contains at least one*

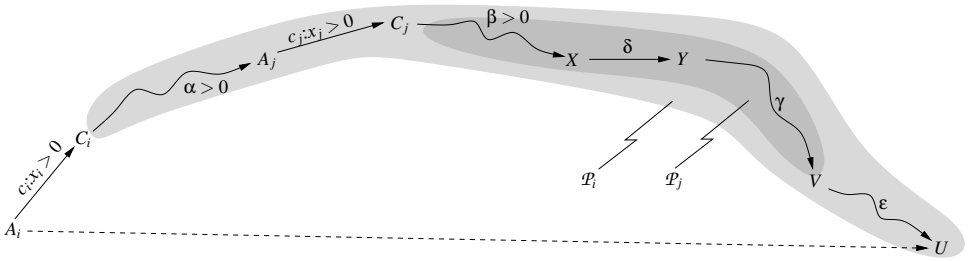

Fig. 7. The scenario addressed by Theorem 1

lower-case edge, let e_j*:* $A_j \xrightarrow{c_j:x_j} C_j$ *be the first lower-case edge that occurs in* $\mathcal{P}_i$*. If* $\mathcal{D}^+(C_i,X) \leq \mathcal{D}^+(C_j,X)$*, then the edge from* A_i *to* U *generated by using* $\mathcal{P}_i$ *to reduce away* e_i *is strictly weaker than an edge (or semi-reducible path) from* A_i *to* U *that can be obtained by bypassing that occurrence of* e_2 *in* $\mathcal{P}$*.*

Proof. First, by Morris' Theorem 3, only breach-free extension sub-paths need be considered [9]. Thus, without loss of generality, $\mathcal{P}_i$ is assumed to be breach-free. That is, $\mathcal{P}_i$ does not contain any occurrences of upper-case edges labeled by C_i. Next, since $\mathcal{D}^+(C_i,X) \leq \mathcal{D}^+(C_j,X)$, there must be a quasi-allowable path from C_i to X of some length $\theta = \mathcal{D}^+(C_i,X) \leq \mathcal{D}^+(C_j,X) \leq \beta < \alpha + x_j + \beta$, where α and β are the (positive) lengths shown in Fig. 7. Finally, let $\mathcal{P}_i'$ be the path obtained by replacing the portion of $\mathcal{P}_i$ from C_i to X by the quasi-allowable path of length θ; and let $\mathcal{P}'$ be the concatenation of the edge e_i and the path $\mathcal{P}_i'$. By construction, $\mathcal{P}'$ is a shorter path than $\mathcal{P}$ and it is breach-free. Thus, the semi-reducibility of $\mathcal{P}$ ensures the semi-reducibility of $\mathcal{P}'$. (Morris used a similar argument in the proof of his Theorem 3.) There are two cases to consider. First, if $\mathcal{P}_i'$ is the extension sub-path for e_i in $\mathcal{P}'$, then using $\mathcal{P}_i'$ to reduce away e_i yields a shorter edge from A_i to U than the edge generated by using $\mathcal{P}_i$. On the other hand, if some proper prefix $\mathcal{P}^\star$ of $\mathcal{P}_i'$ is the extension sub-path for e_i in $\mathcal{P}'$, then the edge generated by using $\mathcal{P}^\star$ to reduce away e_i, followed by the rest of $\mathcal{P}_i'$ is a semi-reducible path from A_i to U that is shorter than the edge generated using $\mathcal{P}_i$.[8] □

Given Theorem 1, it follows that the only semi-reducible paths in $\mathcal{G}'$ that need to be considered by the incremental algorithm are those in which the order of nesting of LC edges is such that the corresponding $\mathcal{D}^+(C_i,X)$ values are *decreasing*—that is, such that the *innermost* LC edges have smaller $\mathcal{D}^+(C_i,X)$ values. Therefore, to ensure that the innermost LC edges are processed first—in *any* relevant path—the DC-checking algorithm need only process LC edges in the order of *non-decreasing* $\mathcal{D}^+(C_i,X)$ values, where X is the source time-point for the edge being inserted into the network.

The *Inky* algorithm assumes that all $\mathcal{D}^+(C_i,T)$ values are available for the DC network *prior* to the insertion of the new edge from X to Y. Thus, the algorithm must ensure that all such values are *updated* so that they will be available for the next invocation.

[8] Any suffix of a breach-free extension sub-path is necessarily semi-reducible [6].

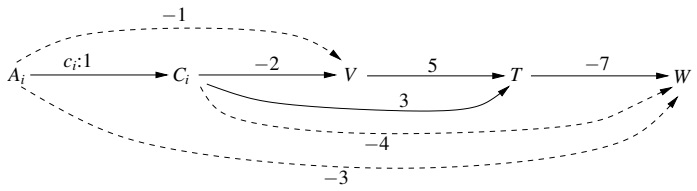

Fig. 8. Alternative ways of reducing away a lower-case edge in an STNU graph

The APPP *matrix.* For each contingent time-point C_i and each time-point T, the *Inky* algorithm also keeps track of whether *all* shortest quasi-allowable paths from C_i to T have the *positive proper prefix* (PPP) property. The reason is that if there is some shortest quasi-allowable path from C_i to T that does *not* have the PPP property, then it is not necessary to generate new edges using allowable paths that contain T, even if such exist. For example, Fig. 8 shows a scenario in which there are two shortest quasi-allowable paths from C_i to T, one having the PPP property and one not. (The single edge from C_i to T has the PPP property; the two-edge path from C_i to V to T does not.) Although a new edge, $A_i \xrightarrow{-3} W$, could be generated by using the shortest allowable path from C_i to T to W to reduce away the lower-case edge, as illustrated on the lower portion of the figure, it is not necessary to do so because the shortest allowable path from C_i to V can be used to reduce away the lower-case edge, generating the new edge, $A_i \xrightarrow{-1} V$, as illustrated on the upper portion of the figure. This edge creates an OU-path from A_i to V to T to W whose length is also -3.

In view of these considerations, the *Inky* algorithm also maintains a K-by-N matrix, called $APPP$, whose values $APPP(C_i,T)$ are all initially *True*. However, if the algorithm ever discovers a shortest quasi-allowable path from C_i to T that does *not* have the PPP property, then it sets $APPP(C_i,T)$ to *False*.

Marking nodes. While searching through the shortest allowable paths emanating from some contingent time-point, C_i, the *Inky* algorithm marks nodes that are the source time-points for any new edges that have been generated during the current invocation. Initially, the only marked node is X (i.e., the source time-point for the new edge being added to the network). If, during the Dijkstra-like traversal of shortest allowable paths, it happens that all nodes remaining in the priority queue are unmarked and have keys (i.e., shortest quasi-allowable path-lengths) that have not changed, then it is certain that no more shorter extension sub-paths can be found; hence, the processing of that lower-case edge can stop. The algorithm uses simple counter variables to keep track of the numbers of marked nodes in the queue and nodes whose keys have changed.

The most-basic form of the Inky *algorithm.* Pseudo-code for the most basic form of the *Inky* algorithm is given in Table 4. Its works as follows. First, it sorts the lower-case edges of $\mathcal{G}$ according to their $\mathcal{D}^+(C_i,X)$ values (Line 1). Next, it creates a boolean vector, called `mark`, that it uses to keep track of the source time-points of any newly created edges (Line 2). Initially, the only new edge is E, the edge to be added to the network. Thus, its source time-point, X, is the only time-point that starts out being marked.

Note that once a time-point is marked, it remains marked for the rest of the invocation of the algorithm. The `globalNumMarks` counter keeps track of how many time-points have been marked. Next, since adding an edge with source time-point X cannot affect a potential function whose sink is X, the initial potential function, f, is *rotated* so that it now uses X as its sink (Line 4). Note that this is done using only the edges from $\mathcal{G}$, ignoring the edge E. This is simply a re-packaging of the `sinkPotSinkDijk` function seen earlier in Section 3.

The main loop of the algorithm, which has exactly K iterations, spans Lines 5 thru 44. The i^{th} iteration processes the contingent link, (A_i, x_i, y_i, C_i), beginning on Line 6. Any edges generated by reducing away the corresponding lower-case edge will be added to the set, `newEdges`, which is initially empty (Line 7). The Dijkstra-like traversal of shortest quasi-allowable paths emanating from C_i uses a priority queue, `Q`, which is initially empty (Line 8). However, each time-point T is immediately inserted into the queue using the corresponding $\mathcal{D}^+(C_i, T)$ value (Line 9). Note that the sink-based potential function, f, is used to convert the path length into the appropriate non-negative value. The $\mathcal{D}^+(C_i, T)$ values, of course, do not reflect the presence of the new edge E, but they make good initial values for the queue. The `numMarksInQueue` and `numChangedInQueue` counters are initialized in Lines 10-11. The `while` loop on Line 12 runs as long as it is possible for some shortest allowable path to be discovered that might be used to generate a meaningful new edge by reducing away the i^{th} lower-case edge. Note that if both counters are ever both zero, it would imply that no path to any time-point remaining in the queue could possibly generate a useful new edge.

At Line 13, the next node T is popped from the queue. Lines 14-15 ensure that the two counters `numMarksInQueue` and `numChangedInQueue` are updated if necessary. Note that the potential function, f, is used to convert the $\mathcal{D}^+(C_i, T)$ value to its corresponding non-negative value, as seen earlier. Also, `T.key` denotes length of the shortest quasi-allowable path from C_i to T that has just been discovered, which is the *key* associated with the time-point T in the queue. Line 16 determines whether the path from C_i to T is an extension sub-path (cf. the definition of extension sub-path in Section 2.3). The length of the new edge from A_i to T that would be generated by using that extension sub-path to reduce away the lower-case edge is computed in Line 17, and stored in the variable `newVal`. Note that because the potential function is based solely on shortest paths in the OU-graph, the adjusted length of the *lower-case* edge (i.e., $f(A_i, x_i, C_i)$) will typically be negative. As a result, `newVal` itself could be negative, which would indicate that a negative semi-reducible loop has been found (i.e., that inserting the edge E into the graph $\mathcal{G}$ made the network *not* dynamically controllable); hence, Line 18 returns `False` if `newVal` is negative. Line 19 then checks whether an edge of length `newVal` would be shorter than any pre-existing edge from A_i to T in the graph. Note that the function, `currAdjEdgeLen`, uses the potential function f to convert the length of any pre-existing edge to its non-negative counterpart. If the new edge would be shorter, it is then added to the set `newEdges` (Line 20).

Lines 21-29 then check each *ordinary* edge emanating from T to determine whether any shortest-path values in the queue can be updated. For each successor edge, $T \xrightarrow{r} W$, the length of the quasi-allowable path from C_i to T to W (using non-negative values) is computed (Line 22) to determine whether the key for W that is currently stored in the

queue needs to be decreased (Lines 23, 25). Line 24 ensures that the `numChangedInQueue` counter is properly updated if the key for W is being changed for the first time. Lines 26-27, still in the case where the key for W was decreased, determine whether the new shortest path from C_i to W has the PPP property and, if so, set APPP(C_i, W) to *True.* In contrast, Lines 28-29, which do not necessarily fall within that case, determine whether a quasi-allowable path from C_i to W has been found that does *not* have the PPP property, in which case, the APPP value is set to *False.*

Lines 30-37 then check each *upper-case* successor edge emanating from T in a similar fashion. Lines 32-33 check whether the resulting path can reduce to an ordinary edge, courtesy of the *Label Removal* rule from Table 1. If so, the path is processed by Lines 22-29, as discussed above. Otherwise, Lines 34-37 determine whether the resulting path can be used to generate a new upper-case edge; if so, the edge is added to `newEdges`. Note that `UC_val` being negative would imply that a negative semi-reducible loop had been found (i.e., that inserting E into the network made it non-DC, Line 35).

At the end of the i^{th} iteration, if `newEdges` is non-empty (Line 38), then several steps are taken to incorporate the newly generated edges. First, in anticipation of adding new edges, each of which has A_i as its source, the potential function is rotated so that it uses A_i as its sink (Line 39). Next, the new edges are added to the graph (Line 40). Since each new edge has A_i as its source, those edges cannot disturb the recently rotated potential function. Finally, Lines 41-43 ensure that if this is the first time that new edges with source time-point A_i are being added to the network, then A_i becomes marked and the `globalNumMarks` counter is updated (Note that A_i might be the activation time-point for multiple contingent links; so this might not be the first time that A_i is marked.)

Finally, if all K iterations of searching for extension sub-paths to generate new edges fail to find a negative loop (cf. Lines 18, 35), then the algorithm returns *True* at Line 45.

Note that in the process of performing the K iterations, the *Inky* algorithm computes all updates of the $\mathcal{D}^+$ matrix that will be needed by subsequent invocations of the algorithm. It also computes updates to the APPP matrix. Furthermore, since the algorithm runs at most K iterations, each of which runs in $O(N^2)$ time (due to the Dijkstra-like traversals), the algorithm runs in $O(N^3)$ time overall.

Improving the Incremental Algorithm. The *Inky* algorithm can be substantially improved as follows. Suppose that C_1 is the first contingent time-point to be processed by the main loop of the algorithm (Lines 5-44). Consider the Dijkstra-like traversal of shortest quasi-allowable paths emanating from C_1. Given that each time-point T is initially inserted into the priority queue, Q, using the corresponding $\mathcal{D}^+(C_1, T)$ value from the previous invocation of the algorithm (Line 9), any time-point that is popped off the queue before X need not have its successor edges processed (cf. Lines 22-29 and 31-37) because following those edges could not possibly change any path-lengths currently stored in the queue. However, once X has been popped off the queue, normal processing of successor edges must resume. For subsequent iterations of the main loop (i.e., when processing other LC edges), a similar approach can be used. In particular, if $\mathcal{A} = \{A_{i_1}, A_{i_2}, \ldots, A_{i_s}\}$ is the set of source time-points for the edges that have been generated so far—all of which must be activation time-points for already-processed LC edges—then until X or some member of $\mathcal{A}$ has been popped off the queue, any other time-point that is popped off the queue need not have its successor edges processed,

because doing so could not possibly change any values stored in the queue. However, once X or some member of $\mathcal{A}$ is popped, then normal processing must be resumed. Given that the worst-case complexity of Dijkstra's algorithm is $O(m + N \log N)$, where m is the number of edges in the graph, reducing the number of successor edges that must be followed during the Dijkstra-like traversals necessarily makes the algorithm more efficient.

A similar technique can be used to make the `rotatePotentialFunc` (Line 4) more efficient. In this case, the algorithm must keep track of the *sink* time-point Y of the new edge E, and the *sink* time-points of any edges that have already been generated by prior iterations of the main loop. As long as none of those time-points have been popped off the queue during the sinkDijkstra traversal used by `rotatePotentialFunc`, the predecessor edges of other time-points that are popped off the queue need not be followed. However, once any of those sink time-points is popped, normal processing must resume. Once again, reducing the number of edges that must be followed during a Dijkstra traversal necessarily makes the *Inky* algorithm more efficient.

Finally, when considering whether to generate a new edge from a given allowable path (Lines 19-20), instead of simply checking whether the new edge would be shorter than any pre-existing edge involving the same time-points, the *Inky* algorithm could use an extra, sourceDijkstra traversal in each iteration of the main loop to compute all $\mathcal{D}^*(A_i, T)$ values, where $\mathcal{D}^*$ is the all-pairs-shortest-semi-reducible-path matrix discussed in Section 2. On the positive side, this could reduce the number of generated edges, leading to additional savings; on the negative side, this technique requires additional computations to maintain the $\mathcal{D}^*(A_i, T)$ values across invocations. Thus, the viability of this technique, unlike the previous two, must be tested empirically.

5 Conclusions

This paper introduced new techniques for speeding up both full and incremental DC checking for STNUs. The *Speedy* algorithm is the full DC-checking algorithm the results from applying these techniques to the Morris-N^4 algorithm. It has been shown to out-perform the Morris-N^4 algorithm, in some cases by an order of magnitude. It is not yet known whether *Speedy* will be competitive with the more recent $O(N^3)$ algorithm presented after the initial publication of the *Speedy* algorithm.

The main contribution of the paper is the *Inky* algorithm, which is the incremental algorithm that results from applying similar techniques to the incremental DC-checking problem. Like its fastest competitors [10, 15], the *Inky* algorithm is $O(N^3)$. However, the techniques it employs to reduce redundant computations, and the fact that its main loop has K iterations, together suggest that *Inky* may out-perform its incremental competitors, each of which can involve up to N iterations. Thus, it is expected that in scenarios where the number of contingent links is relatively small compared to the total number of time-points, *Inky* may perform especially well.

Clearly, a thorough comparative, empirical evaluation of the top incremental DC-checking algorithms is needed. However, given that the competitors have been so recently introduced, such an evaluation must be left to future work.

References

1. Chien, S., Sherwood, R., Rabideau, G., Zetocha, P., Wainwright, R., Klupar, P., Gaasbeck, J.V., Castano, R., Davies, A., Burl, M., Knight, R., Stough, T., Roden, J.: The techsat-21 autonomous space science agent. In: The First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002). pp. 570–577. ACM Press (2002)
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2009)
3. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial Intelligence 49, 61–95 (1991)
4. Hunsberger, L.: Efficient execution of dynamically controllable simple temporal networks with uncertainty, Under review
5. Hunsberger, L.: A faster execution algorithm for dynamically controllable STNUs. In: Proceedings of the 20th Symposium on Temporal Representation and Reasoning (TIME-2013) (2013)
6. Hunsberger, L.: Magic loops in simple temporal networks with uncertainty. In: Proceedings of the Fifth International Conference on Agents and Artificial Intelligence (ICAART-2013) (2013)
7. Hunsberger, L.: A faster algorithm for checking the dynamic controllability of simple temporal networks with uncertainty. In: Proceedings of the 6th International Conference on Agents and Artificial Intelligence (ICAART-2014). SciTePress (2014)
8. Hunsberger, L., Posenato, R., Combi, C.: The dynamic controllability of conditional stns with uncertainty. In: Proceedings of the PlanEx Workshop at ICAPS-2012. pp. 121–128 (2012)
9. Morris, P.: A structural characterization of temporal dynamic controllability. In: Principles and Practice of Constraint Programming (CP 2006), Lecture Notes in Computer Science, vol. 4204, pp. 375–389. Springer (2006)
10. Morris, P.: Dynamic controllability and dispatchability relationships. In: Simonis, H. (ed.) Integration of AI and OR Techniques in Constraint Programming, Lecture Notes in Computer Science, vol. 8451, pp. 464–479. Springer (2014)
11. Morris, P., Muscettola, N., Vidal, T.: Dynamic control of plans with temporal uncertainty. In: Nebel, B. (ed.) 17th International Joint Conference on Artificial Intelligence (IJCAI-01). pp. 494–499. Morgan Kaufmann (2001)
12. Morris, P.H., Muscettola, N.: Temporal dynamic controllability revisited. In: Veloso, M.M., Kambhampati, S. (eds.) The 20th National Conference on Artificial Intelligence (AAAI-2005). pp. 1193–1198. MIT Press (2005)
13. Nilsson, M., Kvarnstrom, J., Doherty, P.: Incremental dynamic controllability revisited. In: Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS-2013) (2013)
14. Nilsson, M., Kvarnstrom, J., Doherty, P.: EfficientIDC: A faster incremental dynamic controllability algorithm. In: Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS-2014) (2014)
15. Nilsson, M., Kvarnstrom, J., Doherty, P.: Incremental dynamic controllability in cubic worst-case time. In: Proceedings of the 21st International Symposium on Temporal Representation and Reasoning (TIME-2014) (2014), forthcoming
16. Shah, J., Stedl, J., Robertson, P., Williams, B.C.: A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In: Boddy, M., Fox, M., Thiébaux, S. (eds.) Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007). AAAI Press (2007)
17. Stedl, J., Williams, B.C.: A fast incremental dynamic controllability algorithm. In: Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check. pp. 69–75 (2005)

Inputs: $\mathcal{G}$, a graph for a dynamically controllable STNU with K cont. links and N time-pts.
$\mathcal{D}^+$, the K-by-N matrix of shortest quasi-allowable path-lengths in $\mathcal{G}$
APPP, the K-by-N matrix of boolean values, discussed in the text
f, a sink-based potential function for $\mathcal{G}$ that uses some time-point S as its sink
E: $X \xrightarrow{\delta} Y$, a new edge to be inserted into the graph $\mathcal{G}$;

Output: `True` if inserting E into $\mathcal{G}$ preserves its dynamic controllability; `False` otherwise.

```
 1. order := lower-case edges of G, sorted into non-decreasing order of D+(Ci,X) values.
 2. mark := boolean vector whose N entries are all initially False, except mark[X] := True.
 3. globalNumMarks := 1;
 4. f := rotatePotentialFunc(G,f,X);  // now X is sink for f
 5. for i = 1 to K:                                                    (Main Loop)
 6.   Let (Ai,xi,yi,Ci) be the i^th contingent link according to order.
 7.   newEdges := {}.
 8.   Q :=  empty priority queue of length N.
 9.   forEach node T in G: insert(Q, T, val), where val = f(Ci,D+(Ci,T),T);
10.   numMarksInQueue := globalNumMarks;
11.   numChangedInQueue := 0;
12.   while ((numMarksInQueue(Q) > 0) && (numChangedInQueue(Q) > 0)):
13.     T := pop(Q); // T is next time-point to be processed
14.     if (mark[T]) numMarksInQueue--;
15.     if (T.key < f(Ci,D+(Ci,T),T)) numChangedInQueue--;
16.     if ((T ≠≡ Ci) && APPP(Ci,T) && (f^-1(Ci,T.key,T) ≤ 0)): // ext. sub-path!
17.       newVal :=  f(Ai,xi,Ci)  + T.key;
18.       if (newVal < 0) return False;
19.       if (newVal < currAdjEdgeLen(f,G,Ai,T)):
20.         newEdges += makeEdge(Ai,f^-1(Ai,newVal,T),T); // generate new edge
21.     forEach ordEdge(T,r,W) in successors(G,T):
22.       len := T.key + f(T,r,W);
23.       if (stillInQueue(Q,W) && (len < W.key)):
24.         if (W.key == f(Ci,D+(Ci,W),W)) numChangedInQueue++;
25.         decreaseKey(Q,W,len);
26.         if (APPP(Ci,T) && ((f^-1(Ci,T.key,T) > 0) || (T ≡ Ci)))
27.           APPP(Ci,W) := True;
28.       if ((len <= W.key) && (T ≠≡ Ci) && (!APPP(Ci,T)||(f(Ci,T.key,T) ≤ 0)))
29.         APPP(Ci,W) := False;
30.     forEach UC_Edge(T,Cj,r,Aj) in UCsuccessors(G,T):
31.       UC_len := T.key + f(Ci,r,Aj);
32.       if (UC_len >= f^-1(-xj)) where xj = lower bound on cont. link for Cj
33.         Process as ordinary path of length UC_len, using Lines 22-29 above
34.       else: UC_val := UC_len + f(Ai,xi,Ci);
35.       if (UC_val < 0) return False;
36.         elseif (UC_val < currAdjUCEdgeLen(f,G,Ai,Cj)):
37.           newEdges += makeUC_Edge(Ai,f^-1(Ai,UC_val,Aj),Cj);
38.   if (newEdges):
39.     f := rotatePotentialFunction(G,f,Ai); // now Ai is sink for f
40.     forEach edge in newEdges:  insertEdge(edge,G);
41.     if (!mark[Ai]):
42.       mark[Ai] := True;
43.       globalNumMarks++;
44. end for i = 1 to K.
45. return True.
```

Table 4. Pseudo-code for the basic version of the *Inky* incremental DC-checking algorithm