A Faster Execution Algorithm for Dynamically Controllable STNUs

Luke Hunsberger
Department of Computer Science
Vassar College
Poughkeepsie, NY 12604 USA
Email: hunsberg@cs.vassar.edu

***Abstract*—A Simple Temporal Network with Uncertainty (STNU) is a data structure for representing and reasoning about temporal constraints where the durations of certain temporal intervals—the contingent links—are only discovered during execution. The most important property of an STNU is whether it is dynamically controllable (DC)—that is, whether there exists a strategy for executing time-points that will guarantee that all constraints will be satisfied no matter how the durations of the contingent links turn out. The fastest DC-checking algorithm reported so far is the $O(N^4)$-time algorithm due to Morris (2006). Hunsberger (2010) presented an $O(N^4)$-time *execution algorithm* for dynamically controllable STNUs, the fastest reported so far. This paper improves upon that algorithm, presenting an $O(N^3)$-time execution algorithm for DC STNUs. The increase in speed is due to more efficient management of the so-called "wait" constraints, which must be removed from the network whenever the corresponding contingent link completes.**

I. Background

Simple Temporal Networks with Uncertainty provide a compact way of representing temporal constraints among activities, where the duration of some of the activities may be uncontrollable—but within known bounds. This section summarizes relevant prior work on temporal networks and the important property of dynamic controllability.

A. Simple Temporal Networks

A Simple Temporal Network (STN) is a pair, $(\mathcal{T}, \mathcal{C})$, where $\mathcal{T}$ is a set of time-point variables—or time-points (TPs)—and $\mathcal{C}$ is a set of binary constraints, each having the form, $Y - X \leq \delta_{xy}$, for some $X, Y \in \mathcal{T}$ and $\delta_{xy} \in \mathbb{R}$ [1]. A pair of constraints, $Y - X \leq b$ and $X - Y \leq -a$, are often abbreviated as $Y - X \in [a, b]$. Typically, one of the time-points in $\mathcal{T}$, called Z, has its value fixed at 0. Binary constraints involving Z are equivalent to unary constraints: $X - Z \leq b$ and $Z - X \leq -a$ are equivalent to $X \in [a, b]$. An STN is *consistent* if there is a set of values for its time-points that satisfy all of its constraints. Such a set of values is called a *solution* for the STN.

Each STN, $(\mathcal{T}, \mathcal{C})$, has a corresponding graph, $(\mathcal{N}, \mathcal{E})$, where the nodes in $\mathcal{N}$ correspond to the time-points in $\mathcal{T}$, and the directed edges in $\mathcal{E}$ correspond to the constraints in $\mathcal{C}$. In particular, each constraint, $Y - X \leq b$ in $\mathcal{C}$, corresponds to an edge, $X \xrightarrow{b} Y$ in $\mathcal{E}$. The *all-pairs, shortest-path* matrix for the graph of an STN is called a *distance matrix*, often denoted by $\mathcal{D}$. For any time-points, X and Y, if $\mathcal{D}(X, Y) = \delta$, then the constraint, $Y - X \leq \delta$, must be satisfied in any solution.

The **Fundamental Theorem of STNs** states that the following are equivalent: (1) an STN $\mathcal{S}$ has a solution; (2) its graph $\mathcal{G}$ has no negative loops; and (3) its distance matrix $\mathcal{D}$ has zeroes down its main diagonal [2].

The time-points other than Z are called *executable.* To *execute* a time-point, X, at some time t, means to assign the value t to X. This is represented by inserting the constraints, $X - Z \leq t$ and $Z - X \leq -t$ into the network. These constraints are equivalent to $X \in [t, t]$ (i.e., $X = t$). The executable time-points in an STN are presumed to be under the control of some agent operating in real time.[1] At any time t, the agent is presumed to be free to execute any previously unexecuted TP. Once executed, a time-point's value is permanently fixed.

B. Collapsing Rigid Components

In a consistent STN, two time-points, X and Y, are said to be *rigidly connected* if $\mathcal{D}(X, Y) + \mathcal{D}(Y, X) = 0$. In such a case, although there may be many choices for the values of X and Y, they are not independent; instead, the value, $Y - X$, is restricted to a constant. For example, suppose $\mathcal{D}(X, Y) = 3$ and $\mathcal{D}(Y, X) = -3$. In this case, X and Y must satisfy both $Y - X \leq 3$ and $X - Y \leq -3$; equivalently, $Y - X = 3$. Thus, although X may take on any value, Y must equal $X + 3$.

More generally, any set, R, of rigidly connected time-points is called a *rigid component.*[2] Many researchers [3], [4], [5] have noted that since the time-points in a rigid component are fixed with respect to one another, the entire rigid component, R, can be effectively represented by a single time-point, as follows. First, let X be any time-point in R; X will be the single representative for the rigid component. Next, the edges in the network that interact with time-points in R must be re-oriented toward X, as follows. Any edge, $Y \xrightarrow{\delta} W$, where $W \in R$, is replaced by the edge, $Y \xrightarrow{\delta+w} X$, where $w = \mathcal{D}(W, X)$. Similarly, any edge, $W \xrightarrow{\gamma} Y$, where $W \in R$, is replaced by the edge, $X \xrightarrow{\gamma-w} Y$, again, where $w = \mathcal{D}(W, X) = -\mathcal{D}(X, W)$. Fig. 1 illustrates the collapsing of a rigid component containing time-points, X, W_1, and W_2.

[1] Agents are not part of the semantics of temporal networks; they are used here for expository convenience.

[2] Being rigidly connected is an equivalence relation; so, the notion of a rigid component is well defined.

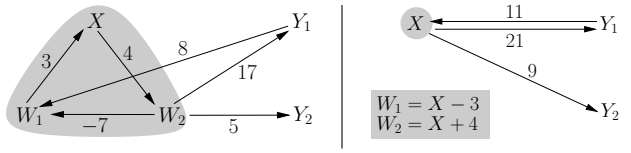

Fig. 1. An STN before (left) and after (right) collapsing a rigid component

C. STNs with Uncertainty

An *STN with Uncertainty* (STNU) is an STN together with a set of *contingent links*, each of which represents a temporal interval whose duration is beyond the control of the planning agent [6]. An STNU is a set $(\mathcal{T}, \mathcal{C}, \mathcal{L})$, where $(\mathcal{T}, \mathcal{C})$ is an STN, and $\mathcal{L}$ is a set of contingent links, each having the form, (A, x, y, C), where $A, C \in \mathcal{T}$, and $0 < x < y < \infty$. For a contingent link, (A, x, y, C), A is called the *activation* time-point, and C is called its *contingent* time-point. Contingent links cannot share contingent time-points; thus, if (A_1, x_1, y_1, C_1) and (A_2, x_2, y_2, C_2) are two contingent links, then C_1 and C_2 must be distinct time-points. However, A_1 and A_2 need not be distinct. Furthermore, the contingent time-point from one contingent link can serve as an activation time-point for another. Thus, contingent links can form chains or trees, but not cycles.[3] In this paper, N denotes the number of time-points in an STNU, and K the number of *contingent* time-points.

D. Dynamic Controllability

The time-points in an STNU are partitioned into three sets: $\mathcal{T}_c$, the contingent time-points; $\{Z\}$, the zero time-point; and $\mathcal{T}_{ex}$, the executable time-points. The agent is presumed to directly control the execution of the executable time-points in an STNU; however, the execution of the contingent time-points is presumed to be beyond the agent's direct control. For example, if (A, x, y, C) is a contingent link for which A is an executable time-point, the agent directly controls only the execution of A. Once A has been executed (i.e., once the contingent link has been *activated*), the execution of C is out of the agent's control. Although C is guaranteed to be executed such that $C - A \in [x, y]$, the agent does not get to choose the particular time, but only *observes* the execution of C when it happens. Similar remarks apply to a tree of contingent links with an executable time-point at its root.

An STNU is called *dynamically controllable* (DC) if there exists a dynamic strategy for executing the executable time-points that guarantees the consistency of the network, no matter how the durations of the contingent links turn out—within their specified bounds. Crucially, the decisions constituting a *dynamic execution stratgey* cannot depend on advance knowledge of the durations of contingent links. Morris, Muscettola and Vidal—hereinafter MMV—presented a concise semantics for dynamic controllability [6]. Hunsberger fixed a technical flaw in their semantics to properly capture the prohibition against decisions based on advance knowledge, and to enable a characterization of dynamic execution strategies in terms of *real-time execution decisions* (RTEDs) [7].

[3]A cycle of contingent links would be inherently inconsistent.

Since RTEDs play a significant role in this paper, they are described in more detail below. RTEDs have two forms: `WAIT` and (t, χ), where $t \in \mathbb{R}$, and $\chi \subseteq \mathcal{T}_{ex}$.

- A `WAIT` decision can be glossed as "Wait for some contingent time-point to execute".
- A (t, χ) decision can be glossed as: "If nothing happens before time t (i.e., if no contingent time-points happen to execute before time t), then execute the (executable) time-points in χ at time t.

The *outcome* of an RTED specifies the time-points that execute during the *next* execution event. Since it frequently depends on the uncertain execution of contingent time-points, the agent typically does not know which outcome will actually happen.

A `WAIT` decision is only applicable if there are some contingent links that are currently activated, but not yet completed. The outcome of a `WAIT` decision specifies the contingent time-point(s) that execute *next.* In contrast, for a (t, χ) decision, there are several possibilities. First, if there are no activated contingent links, then the outcome is completely determined: at time t, the (executable) time-points in χ will be executed. However, if there are some activated contingent links, it may be that one or more contingent time-points happen to execute at some time $\rho < t$. In that case, the outcome involves only the execution of those contingent time-points, not the time-points in χ. A third case, which is extremely rare in practice, involves the possibility that one or more contingent time-points happen to execute exactly at time t. In that case, the outcome involves the simultaneous execution of those contingent time-points together with the time-points in χ.

After observing whichever outcome happens to occur, the agent can then generate a new execution decision. Crucially, that new decision can depend on the information gleaned from the just observed outcome. For example, if an agent's initial decision is $(10, \{X\})$, but some contingent time-point C happens to execute at time 4, then the agent need not remain committed to executing X at time 10. Instead, the agent might decide to execute X at time 7. Furthermore, the agent might decide to execute another time-point, Y, along with X. Thus, the agent's next decision might be $(7, \{X, Y\})$.

E. DC-Checking Algorithms

A *DC-checking algorithm* is an algorithm that determines whether any given STNU is dynamically controllable. Several DC-checking algorithms have been presented in the literature.

1) The MMV DC-Checking Algorithm: MMV presented a *pseudo-polynomial* DC-checking algorithm based on the generation and propagation of a new kind of constraint, called a *wait* [6]. Each wait has the form, *as long as the contingent time-point, C, is unexecuted, B must wait at least w units after the execution of C's activation time-point, A.* The MMV DC-checking algorithm is sound and complete with respect to the corrected MMV semantics [7].

2) The MM DC-Checking Algorithm: Morris and Muscettola—hereinafter MM—presented the first truly polynomial DC-checking algorithm [8]. Given an STNU, $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$, it begins by creating a graph, $\mathcal{G}$, that contains

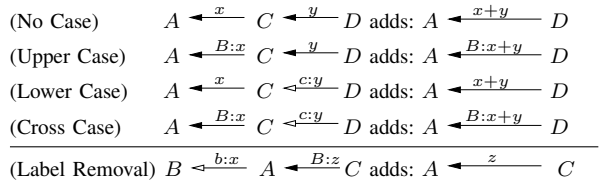

Fig. 2. The edge-generation rules for the MM DC-checking algorithm

all of the edges from the graph of the related STN, $(\mathcal{T}, \mathcal{C})$, plus two new kinds of *labeled edges*. In particular, for each contingent link, (A, x, y, C), $\mathcal{G}$ includes the *lower-case* edge, $A \xrightarrow{c:x} C$, and the *upper-case* edge, $A \xleftarrow{C:-y} C$. The lower-case edge represents the uncontrollable possibility that the duration of the contingent link might be its *minimum* value, x; the upper-case edge represents the uncontrollable possibility that the duration might be its *maximum* value, y. Without their labels, these edges would be mutually inconsistent; thus, the labels must be carefully maintained when propagating constraints (equiv., generating new edges).

For contrast purposes, unlabeled edges, like those in any STN, are called *ordinary* edges. For each contingent link, (A, x, y, C), the STNU graph contains not only the labeled edges described above, but also the following *ordinary* edges: $A \xrightarrow{y} C$ and $A \xleftarrow{-x} C$. Together, these ordinary edges represent the *known fact* that the duration of the contingent link will belong to the interval $[x, y]$.[4]

To generate new edges, the MM DC-checking algorithm uses the rules in Fig. 2, which are equivalent to, but more uniformly and concisely expressed than those used by the MMV algorithm.[5] Note that none of the rules generate new lower-case edges. Thus, an STNU with K contingent links invariably has exactly K lower-case edges. In contrast, the Upper-Case and Cross-Case rules may generate new upper-case edges. However, since the target of an upper-case edge is invariably the activation time-point for the corresponding contingent link, an STNU with K contingent links and N time-points can have at most KN upper-case edges. MM showed that the upper-case edges generated by their algorithm are equivalent to the *waits* generated by the MMV algorithm.

After each round of edge generation, the MM algorithm performs a consistency check on an associated STN, called the *AllMax* STN. Given an STNU, $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$, the *AllMax* STN for $\mathcal{S}$ is the STN—not STNU—that results from forcing all of the contingent links in $\mathcal{L}$ to take on their maximum values. Thus, the *AllMax* STN includes the constraints in $\mathcal{C}$, together with constraints of the form, $C - A = y$, for each contingent link, (A, x, y, C). The *AllMax* STN for $\mathcal{S}$ is denoted by $\mathcal{S}_x$; the graph for the *AllMax* STN is denoted by $\mathcal{G}_x$; and the corresponding distance matrix is denoted by $\mathcal{D}_x$.

Each new ordinary edge generated by the MM algorithm is inserted directly into the *AllMax* graph, $\mathcal{G}_x$. Each new upper-case edge is, first, stripped of its upper-case label and, then, inserted—as an ordinary edge—into the *AllMax* graph. After each round of edge generation, the MM algorithm re-computes the *AllMax* distance matrix, $\mathcal{D}_x$. They proved that, for dynamically controllable networks, no more than $N^2 + NK + K$ rounds of edge generation are required to generate *all* of the derivable edges. They also proved that if $\mathcal{D}_x$ remains consistent after $N^2 + NK + K$ such rounds, then the network must be dynamically controllable. Since each round takes $O(N^3)$ time, the MM algorithm runs in $O(N^5)$ time. The MM DC-checking algorithm is sound and complete.

3) Morris' DC-Checking Algorithm: Morris [9] developed a faster, $O(N^4)$-time DC-checking algorithm by focusing on the edges that could be generated from the fixed supply of lower-case edges in an STNU. In particular, he showed how to more efficiently search for relevant applications of the Lower-Case and Cross-Case rules to each lower-case edge. Morris' DC-checking algorithm performs only K rounds of edge generation using these two rules and propagating the corresponding constraints through the *AllMax* matrix, $\mathcal{D}_x$. He proved that if $\mathcal{D}_x$ were still consistent at the end of this process, then the STNU must be dynamically controllable.

The correctness of Morris' algorithm depends on several important properties. Since these properties will be needed later on, they are summarized below.

a) Path transformations: The rules from Fig. 2 can be viewed as path-transformation rules, as follows. Suppose a path $\mathcal{P}$ contains consecutive edges, e_1 and e_2, to which one of the first four edge-generation rules applies, yielding a new edge, e. The path obtained from $\mathcal{P}$ by replacing e_1 and e_2 by e is called a *transformation* of $\mathcal{P}$. The Label-Removal rule can similarly be used to transform (or *reduce*) a path.

b) Reduced distance: The *reduced distance* of a path in an STNU graph is the sum of the edge lengths in the path, ignoring any labels. Note that the edge-generation/path-transformation rules from Fig. 2 preserve reduced distance.

c) Semi-reducible paths: A path in an STNU graph is called *semi-reducible* if it can be transformed into a path without any *lower-case* edges. For any STNU, the all-pairs-shortest-*semi-reducible*-paths (APSSRP) matrix is denoted by $\mathcal{D}^*$. Thus, for any time-points X and Y, $\mathcal{D}^*(X, Y)$ equals the length of the shortest semi-reducible path from X to Y.

Morris proved that an STNU is dynamically controllable if and only if its graph does not have a semi-reducible negative loop (i.e., a semi-reducible path that forms a loop whose reduced distance is negative). Hunsberger [10] proved that for dynamically controllable STNUs, the *AllMax* matrix computed by the MM algorithm is precisely the APSSRP matrix, $\mathcal{D}^*$. Thus, it follows that an STNU is DC if and only if $\mathcal{D}^*$ has zeroes down its main diagonal. These results collectively constitute the **Fundamental Theorem of STNUs** [2].

Based on an in-depth analysis of the graphical structure

[4]These extra ordinary edges have been shown to be irrelevant to the DC-checking problem [2]. They are retained here for consistency with prior work.

[5]The rules are shown using MM's notation. Note that the x's and y's here are not necessarily bounds for contingent links. Also, 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. In addition, in the Upper-Case and Cross-Case rules, B is contingent, with activation time-point A. The Lower Case rule only applies if $x \leq 0$ and $A \neq C$; the Cross Case rule only applies when $x \leq 0$ and $B \neq C$; the Label-Removal applies when $z \geq -x$.

of semi-reducible paths, Morris' DC-checking algorithm performs at most K rounds of edge generation, looking for potential applications of the Lower-Case or Cross-Case rules. As in the MM algorithm, any new edges that are generated are inserted—without their alphabetic labels—into the *AllMax* graph. If, after inserting all of the edges generated by K such rounds, the *AllMax* matrix, $\mathcal{D}_x$, is still consistent, then the network is necessarily dynamically controllable. Hunsberger [10] proved that for dynamically controllable SNTUs, the *AllMax* matrices computed by the Morris and MM algorithms are both equal to the APSSRP matrix, $\mathcal{D}^*$. This result is particularly interesting since Morris' algorithm, with its focus on the Lower-Case and Upper-Case rules, does *not* typically generate all of the upper-case edges. However, the ones that it does generate are enough to complete the computation of $\mathcal{D}^*$.

F. DC-Checking vs. Execution

A DC-checking algorithm is only responsible for determining whether an STNU is dynamically controllable. That is, it need only ensure the *existence* of a dynamic execution strategy (or, equivalently, an RTED-based strategy); it need not construct such a strategy. However, in successful instances (i.e., when the network in question turns out to be DC), both the MMV and MM DC-checking algorithms generate *all* of the edges that are derivable from the edge-generation rules. Thus, both of these DC-checking algorithms also effectively prepare the network for execution. MMV showed that this full complement of edges can be used as the basis for a real-time execution algorithm—henceforth called the MMV-EX algorithm—that guarantees the consistency of the STNU no matter how the contingent durations turn out. Unfortunately, the MM DC-checking algorithm takes $O(N^5)$ time to generate the information used by the MMV-EX algorithm.

In contrast, Morris' faster, $O(N^4)$-time DC-checking algorithm typically does *not* generate all of the edges that are derivable from the edge-generation rules. Instead, it focuses on "reducing away" the lower-case edges. Thus, for any STNU that passes Morris' algorithm, it seemed plausible that extra work might be required to prepare the network for execution. Toward that end, Morris briefly sketched an extra $O(N^4)$-time procedure to generate the rest of the derivable upper-case and ordinary edges—in advance of execution. The next section demonstrates that this is not necessary.

II. The NEW-EX Execution Algorithm

Hunsberger [10] presented an execution algorithm, called NEW-EX, that takes as its starting point an STNU that has passed Morris' DC-checking algorithm. The NEW-EX algorithm guarantees the consistency of the network throughout the execution of all of its time-points, no matter how the contingent durations turn out—as long as they are within their specified bounds. The main insight behind the algorithm is that the *AllMax* graph, $\mathcal{G}_x$, which does not distinguish ordinary and upper-case edges, contains all of the information needed to generate execution decisions. The reason for this is that, prior to the execution of a contingent time-point C, any upper-case edge, $Y \xrightarrow{C:-w} A$, labeled by C, is indistinguishable from the corresponding ordinary edge, $Y \xrightarrow{-w} A$, in terms of its effect on as-yet-unexecuted time-points. The catch, however, is that after C executes, the conditional "wait" constraints represented by upper-case edges labeled by C are no longer applicable and, thus, must be *removed* from the STNU graph, thereby leading to changes in the *AllMax* graph and its corresponding distance matrix, $\mathcal{D}_x$.[6]

The NEW-EX algorithm is iterative. Each iteration involves the following steps. First, the *AllMax* matrix, $\mathcal{D}_x$, is used to generate an execution decision. Second, the execution outcome is observed: one or more time-points executing at some time t. Third, the STNU and *AllMax* graphs are updated. In particular, for each (contingent or executable) time-point, X, that executed at time t, the following constraints are inserted:

- $X - Z \leq t$ and $Z - X \leq -t$ (i.e., $X = t$)—called *execution* constraints; and
- for each Y that remains unexecuted, $Z - Y \leq -t$ (i.e., $Y \geq t$)—called *greater-than-now* constraints.

In addition, for any *contingent* time-points, C, that might have executed at t, all upper-case edges labeled by C are *removed* from the STNU graph; and their *unlabeled counterparts* are removed from the *AllMax* graph. Finally, the *AllMax* matrix, $\mathcal{D}_x$, is updated in preparation for the next iteration.

Since it costs $O(N^3)$ time per iteration, the updating of the *AllMax* matrix, $\mathcal{D}_x$, is the driving factor behind the $O(N^4)$-time complexity of the NEW-EX algorithm. The rest of this section discusses features of the NEW-EX algorithm, emphasizing those that will be needed in subsequent sections.

A. Core Edges and Related Graphs and Matrices

Let $\mathcal{S} = (\mathcal{T}, \mathcal{C}, \mathcal{L})$ be an STNU and $\mathcal{G}$ its corresponding graph, which includes all the ordinary edges from $\mathcal{C}$, together with the lower-case, upper-case and ordinary edges associated with the contingent links in $\mathcal{L}$. The *core edges* for $\mathcal{S}$ are all of the ordinary and upper-case edges from $\mathcal{G}$, together with all of the edges generated by Morris' DC-checking algorithm. Note that the core edges do *not* include any lower-case edges. In addition, the core edges, stripped of any alphabetic labels, are precisely the (ordinary) edges that appear in the *AllMax* graph, $\mathcal{G}_x$, at the end of Morris' DC-checking algorithm.

For quick access, the NEW-EX algorithm stores the *upper-case* core edges in a K-by-N matrix, called `UC`. In particular, if $Y \xrightarrow{C:-w} A$ is the strongest core upper-case edge from Y to A labeled by C, then $\texttt{UC}(C, Y) = -w$.[7]

The NEW-EX algorithm also uses an STN graph, $\mathcal{G}_o$, that initially consists of all the *ordinary* core edges from $\mathcal{S}$. The corresponding distance matrix is denoted by $\mathcal{D}_o$. During execution, the NEW-EX algorithm adds constraints to both $\mathcal{G}_x$ and $\mathcal{G}_o$, and updates the corresponding matrices, $\mathcal{D}_x$ and $\mathcal{D}_o$.

[6] As a consequence, during execution, the *AllMax* graph only forces all as-yet-*unexecuted* contingent links to take on their maximum durations.

[7] C is used as an index into the `UC` matrix instead of A, since multiple contingent links could have the same activation time-point, A.

B. Removing Upper-Case Edges during Execution

As already mentioned, whenever any contingent time-point C happens to execute, the upper-case edges labeled by C are no longer relevant and, thus, must be removed from the STNU graph.[8] Doing so involves *removing* or *weakening* edges in the graph, $\mathcal{G}_x$, which, in turn, requires updating the matrix, $\mathcal{D}_x$. In particular, during execution, $\mathcal{D}_x$ must be properly updated so that its entries reflect only that which can be derived from the core upper-case edges associated with *unexecuted* contingent time-points, as well as all core ordinary edges.[9]

Prior to observing the next execution event, the NEW-EX algorithm prepares a helper matrix, $\mathcal{D}'$, that will only be needed should a contingent time-point happen to execute. First, $\mathcal{D}'$ is initialized to $\mathcal{D}_o$, effectively representing the removal of *all* upper-case edges from $\mathcal{D}_x$. Second, since *unactivated* contingent time-points cannot be part of the next execution event, the core upper-case edges corresponding to *unactivated* contingent time-points are copied from the UC matrix into $\mathcal{D}'$. Third, the $O(N^3)$-time Floyd-Warshall algorithm [11] is used to fully propagate these changes to $\mathcal{D}'$. Later on, if the execution outcome is observed to include the execution of one or more contingent time-points, the NEW-EX algorithm continues the updating of $\mathcal{D}'$ to include the core upper-case edges for all remaining *unexecuted* contingent time-points, and sets $\mathcal{D}_x$ to $\mathcal{D}'$. Thus, the NEW-EX algorithm deals with the removal of upper-case edges for executed contingent time-points by, in effect, removing *all* upper-case edges and, then, re-inserting those corresponding to unexecuted contingent time-points. The FAST-EX algorithm, introduced in the next section, uses a completely different technique to achieve an order-of-magnitude speed-up in this process.

III. FAST-EX: A Faster Execution Algorithm

This section presents the FAST-EX algorithm, a faster, $O(N^3)$-time execution algorithm for dynamically controllable STNUs. The FAST-EX algorithm achieves its better performance by carrying out the updating of the *AllMax* matrix more efficiently, taking only $O(N^2)$ time per iteration. To do this, the FAST-EX algorithm uses the following techniques:

- It focuses on $\mathcal{D}_x$ entries involving Z, which are the only entries that are needed to generate execution decisions.
- It collapses down to a single point the *rigid component* consisting of Z and all already-executed time-points.
- Similarly to Johnson's algorithm [11], it uses a *potential function* to convert all edge-weights to non-negative values, thereby enabling Dijkstra's *single-source shortest-paths* (SSSP) algorithm [11] to be used to update the needed $\mathcal{D}_x$ entries.

Each of these techniques is discussed in more detail below.

Let $\mathcal{U}_x$ = set of as-yet-unexecuted non-contingent time-points
IF $\mathcal{U}_x$ is empty,
THEN $RD :=$ WAIT
ELSE $RD := (t, \chi)$, where:
$t := \min\{-\mathcal{D}_x(X, Z) \mid X \in \mathcal{U}_x\}$
$\chi := \{X \in \mathcal{U}_x \mid -\mathcal{D}_x(X, Z) = t\}$

Fig. 3. Pseudo-code for generating the next execution decision

A. Restricting Attention to $\mathcal{D}_x$ Entries involving Z

Like the NEW-EX algorithm, the FAST-EX algorithm uses the method shown in Fig. 3 to generate the next real-time execution decision [10].[10] Clearly, the execution decisions generated in this way depend only on $\mathcal{D}_x$ entries involving Z. Therefore, it is not necessary for an execution algorithm to maintain the entire $\mathcal{D}_x$ matrix—as is done by NEW-EX. Instead, it suffices to maintain only the values involving Z. Thus, for each unexecuted time-point X, the FAST-EX algorithm only maintains the values $\mathcal{D}_x(X, Z)$ and $\mathcal{D}_x(Z, X)$.

B. Using Dijkstra's Algorithm to Update $\mathcal{D}_x$

As will be seen, every edge that the FAST-EX algorithm adds to or removes from the *AllMax* graph during execution necessarily involves Z. This special feature enables a more efficient way of updating the desired distance-matrix entries. The technique relies on the following theorem.

Theorem 1: Suppose that X and Z are distinct time-points in an STN, and that $\mathcal{P}$ is a shortest path from X to Z. Then there exists a shortest path from X to Z that does not include any edges of the form, $Z \xrightarrow{\delta} Y$.

Proof: Suppose X and Z are distinct time-points in an STN, and that $\mathcal{P}$ is a shortest path from X to Z. Suppose further that $\mathcal{P}$ includes an edge, E, of the form, $Z \xrightarrow{\delta} Y$. In that case, the suffix of $\mathcal{P}$ whose first edge is E is a loop from Z to Z. Since any sub-path of a shortest path is necessarily itself a shortest path [5], it follows that the length of this loop is 0. Thus, extracting this loop from $\mathcal{P}$ yields a new path, $\mathcal{P}'$, from X to Z whose length is the same as the length of $\mathcal{P}$. This process can be repeated until the resulting shortest path does not contain any edges of the form, $Z \xrightarrow{\delta} Y$. ■

Corollary 1: Suppose X and Z are distinct time-points in an STN, and that $\mathcal{P}$ is a shortest path from Z to X. Then there exists a shortest path from Z to X that does not include any edges of the form, $Y \xrightarrow{\delta} Z$.

In view of these results, it follows that whenever an edge of the form, $Y \xrightarrow{\delta} Z$, is added to (or removed from) an STN graph, $\mathcal{G}$, it cannot affect the lengths of shortest paths whose source time-point is Z. In other words, adding such an edge to

[8] *Non-core* upper-case edges are not explicitly represented in the STNU graph; they only appear implicitly as paths terminating in *core* upper-case edges. Thus, only core upper-case edges need be removed during execution.

[9] Since Morris' DC-checking algorithm extracts all meaningful constraints from lower-case edges, the NEW-EX algorithm ignores lower-case edges.

[10] Because the NEW-EX algorithm uses a technique of splitting the zero time-point, Z, into two separate time-points, $\mathtt{Z_{in}}$ and $\mathtt{Z_{out}}$—a technique which is *not* used by the FAST-EX algorithm—the description of the procedure for generating execution decisions uses expressions such as $\mathcal{D}_x(Z, X)$ and $\mathcal{D}_x(X, Z)$, instead of $\mathcal{D}_x(\mathtt{Z_{out}}, X)$ and $\mathcal{D}_x(X, \mathtt{Z_{in}})$. This difference is not important in the context of generating execution decisions.

INPUTS:

$\mathcal{G}$, an STN graph

$\mathcal{D}$, the distance matrix for $\mathcal{G}$, except that only the entries of the form, $\mathcal{D}(Z, X)$, for all X, are guaranteed to be correct.

OUTPUT:

$\mathcal{D}$, updated so that entries, $\mathcal{D}(X, Z)$, for all X, are correct.

(1) For each X, let $h(X) = \mathcal{D}(Z, X)$.
(2) Create a new graph, $\mathcal{G}^\star$, whose edges correspond to those in $\mathcal{G}$, as follows. For each edge, $U \xrightarrow{w} V$, in $\mathcal{G}$, there is an edge, $U \xrightarrow{h(U)+w-h(V)} V$, in $\mathcal{G}^\star$.
(3) Run Dijkstra's SSSP Algorithm on $\mathcal{G}^\star$, using Z as the single sink.
(4) For each X, set $\mathcal{D}(X, Z) = \mathcal{D}^\star(X, Z) - h(X)$.

Fig. 4. Pseudo-code for the *SinkDijkstra* algorithm

an STN (or removing it from the STN) cannot cause changes to distance-matrix entries of the form, $\mathcal{D}(Z, X)$. Thus, as in Johnson's algorithm [11], the values, $h(X) = \mathcal{D}(Z, X)$, can be used as a *potential function* to re-write the edge-weights in the graph so that they will all be non-negative. To see this, note that for any edge, $U \xrightarrow{w} V$, $h(V) \leq h(U) + w$, since the length of the shortest path from Z to V must be less than or equal to the length of the shortest path from Z to V via U. But then $h(U) + w - h(V) \geq 0$. Thus, a new graph, $\mathcal{G}^\star$, is created containing edges derived from those in the original graph, $\mathcal{G}$. In particular, each edge, $U \xrightarrow{w} V$, in $\mathcal{G}$ gives rise to an edge, $U \xrightarrow{h(U)+w-h(V)} V$, in $\mathcal{G}^\star$.

Next, since all edge-weights in $\mathcal{G}^\star$ are non-negative, Dijkstra's single-*sink*-shortest-paths (SSSP) algorithm can be run using Z as the single sink/destination. Shortest path information in the new graph is easily translated into shortest path information in the original graph, as follows. For each time-point X, $\mathcal{D}(X, Z) = \mathcal{D}^\star(X, Z) - h(X)$, where $\mathcal{D}^\star$ is the distance matrix for the new graph $\mathcal{G}^\star$. (Of course, Dijkstra's algorithm does not compute *all* of the entries in $\mathcal{D}^\star$; it only computes those terminating in Z.)

For convenience, the technique just described is called *SinkDijkstra*, since it computes the lengths of shortest paths whose sink (or destination) is Z. Pseudo-code for the *SinkDijkstra* procedure is given in Fig. 4.

Similarly, whenever an edge of the form, $Z \xrightarrow{\delta} Y$, is added to (or removed from) an STN, it cannot affect the lengths of the shortest paths whose destination time-point is Z. That is, it cannot affect any entry of the form, $\mathcal{D}(X, Z)$. Thus, $h(X) = \mathcal{D}(X, Z)$ can be used as a potential function to re-write the edge-weights, to make them all non-negative, and Dijkstra's single-*source*-shortest-paths (SSSP) algorithm can be run on the new graph to generate all of the updated $\mathcal{D}(Z, X)$ values. This procedure, which is analogous to *SinkDijkstra,* is called *SourceDijkstra,* since it uses Z as its single source.

Since the only kinds of edges that are added to (or removed from) the *AllMax* graph, $\mathcal{G}_x$, during the execution of the time-points in an STNU are edges that involve Z, the FAST-EX algorithm uses *SinkDijkstra* and *SourceDijkstra*, in alternation, to compute the needed updates to the *AllMax* matrix, $\mathcal{D}_x$.

Like the NEW-EX algorithm, whenever a time-point, X, is executed at some time t, the FAST-EX algorithm adds the

(1) Add the constraint, $X - Z \leq t$ (i.e., $X \leq t$), to the graph $\mathcal{G}_x$. Then run *SourceDijkstra* to generate updated entries of the form, $\mathcal{D}_x(Z, W)$, for each time-point W.
(2) Add the constraint, $Z - X \leq -t$ (i.e., $X \geq t$), and, for each as-yet-unexecuted time-point Y, add the *greater-than-now* constraint, $Z - Y \leq -t$ (i.e., $Y \geq t$). Then run *SinkDijkstra* to generate updated entries of the form, $\mathcal{D}_x(W, Z)$, for each time-point W.

Fig. 5. FAST-EX's response to the execution of X at time t

execution constraint, $X = t$, to $\mathcal{G}_x$. In addition, for each as-yet-unexecuted time-point, Y, it adds the *greater-than-now* constraint, $Y \geq t$. However, FAST-EX must perform these constraint insertions in a particular order to ensure that it can use the *SinkDijkstra* and *SourceDijkstra* procedures. The FAST-EX algorithm takes the steps shown in Fig. 5.

C. Managing the Rigid Component of Executed Time-Points

At any time during execution, let $\mathcal{R}$ denote the set consisting of Z together with all of the already-executed time-points. Since the execution of each time-point causes it to become rigid with Z, the set $\mathcal{R}$ is a rigid component. Using the techniques described in Section I, the FAST-EX algorithm represents the rigid component, $\mathcal{R}$, by a single point. For convenience, Z is chosen as the representative time-point for $\mathcal{R}$. As each time-point, X, executes, it joins $\mathcal{R}$ and is, thus, effectively removed from the network. In particular, edges that formerly pointed to X are re-directed toward Z; and edges that formerly emanated from X are re-directed to emanate from Z.

The FAST-EX algorithm uses multiple hash tables [11] to store the edges that belong to the *AllMax* graph. In particular, for each time-point X, the hash table, $Ins(X)$, stores all of the edges in the *AllMax* graph that point at X (i.e., that have X as their destination). For example, an edge, $Y \xrightarrow{\delta} X$, would be stored in the hash table $Ins(X)$ with a key of Y, and a value of δ. Similarly, all of the edges in the *AllMax* graph that emanate from X (i.e., that have X as their source) are stored in a hash table, $Outs(X)$. Although each edge is stored in two hash tables, this bit of redundancy enables fast access.

Now, when a time-point X is executed at some time t, any edge whose destination is X must be moved from $Ins(X)$ to $Ins(Z)$, with its weight appropriately adjusted. Similarly, any edge whose source is X must be moved from $Outs(X)$ to $Outs(Z)$. These kinds of changes are also made to the FAST-EX algorithm's store of *ordinary* core edges, as follows.

Prior to execution, the FAST-EX algorithm stores the *ordinary* core edges resulting from Morris' DC-checking algorithm in an N-by-N matrix, called `OC`. In particular, for any time-points, X and Y, the entry, $\texttt{OC}(X, Y)$, equals the length of the strongest ordinary core edge from X to Y resulting from Morris' algorithm.[11] (If no such edge exists, then $\texttt{OC}(X, Y) = \infty$.) Because the FAST-EX algorithm re-directs edges in the process of collapsing the rigid component, $\mathcal{R}$, the entries in the `OC` matrix are similarly re-directed. For example,

[11] Because the NEW-EX algorithm carries along an extra distance matrix, $\mathcal{D}_o$, that propagates all ordinary constraints, whether core or those arising from execution, the NEW-EX algorithm does not need the `OC` matrix.

suppose that $\texttt{OC}(X, Y) = 22$ is an initial entry in the OC matrix, representing a core ordinary edge, $X \xrightarrow{22} Y$. Now suppose that X is subsequently executed at time 5. Redirecting the above edge to emanate from Z yields the edge, $Z \xrightarrow{27} Y$. If the current entry, $\texttt{OC}(Z, Y)$, is greater than 27, representing a weaker constraint, it must be strengthened: $\texttt{OC}(Z, Y) := 27$. If the current entry is less than or equal to 27, representing a stronger constraint, no change to $\texttt{OC}(Z, Y)$ is made. Similar remarks apply to re-directing entries of the form, $\texttt{OC}(Y, X)$. Note that no additional propagation is done during this re-direction process.

In contrast, the FAST-EX algorithm does not do any re-directing of entries in the UC matrix. Implications of this minor point are addressed in the next section.

D. Dealing with the Removal of Upper-Case Edges

The main contribution of the FAST-EX algorithm is its more efficient processing of the *removal* (or *weakening*) of edges from the *AllMax* graph, $\mathcal{G}_x$, that occurs whenever a contingent time-point is executed. Recall that for each contingent link, (A, x, y, C), the upper-case edges labeled by C (available in the UC matrix) are stored—without any alphabetic labels—in the *AllMax* graph, using the Ins and $Outs$ hash tables.[12] (Being an STN graph, the *AllMax* graph cannot distinguish ordinary and upper-case edges.) Now, before a contingent time-point C can execute, its activation time-point, A, must have already executed. Thus, A must have already joined the rigid component, $\mathcal{R}$, and all edges in the *AllMax* graph (i.e., in the Ins and $Outs$ hash tables) that involve A must have already been re-directed to involve Z. Furthermore, since each upper-case edge labeled by C necessarily points at A (in the STNU graph), each edge in the *AllMax* graph that derives from an upper-case edge labeled by C must have already been re-directed to point at Z. Thus, removing the core upper-case edges labeled by C from the STNU graph corresponds to removing (ordinary) edges from the *AllMax* graph that point at Z. This paves the way for the use of the *SinkDijkstra* procedure to compute the necessary updates to $\mathcal{D}_x$. However, before that can be done, the FAST-EX algorithm must deal with the possibility that the removal of an edge from some Y to Z in the *AllMax* graph might effectively *uncover* some previously longer edge that, due to the first edge's removal, now becomes the shortest edge from Y to Z.

Consider the following example. Suppose that, due to the contingent time-point C having just executed, an upper-case edge, $Y \xrightarrow{C:-9} A$, is to be removed from the STNU graph. Now, because A must have executed previously—say, at time 8—this edge, stripped of its label, might be in the *AllMax* graph as an ordinary edge from Y to Z of length -17. Removing this edge from the *AllMax* graph is necessary. But what should it be replaced by? Well, an entry, $\texttt{OC}(Y, Z) = -14$, in the matrix of core ordinary edges would give rise to an edge, $Y \xrightarrow{-14} Z$, in the *AllMax* graph. Alternatively, any upper-case edge of the form, $Y \xrightarrow{C_i:-w_i} A_i$, for which the activation time-point A_i has already executed, but the contingent time-point C_i has not, also gives rise to an ordinary edge from Y to Z in the *AllMax* graph. (The length of that edge is $-w_i - a_i$, where a_i is the time at which A_i executed.) Whichever of these edges leads to the shortest edge from Y to Z in the *AllMax* graph is the one that needs to be inserted into the Ins and $Outs$ hash tables as a replacement for the edge that was removed. The relevant entries are $\texttt{UC}(C_i, Y)$ for each activated-but-unexecuted contingent time-point C_i, as well as $\texttt{OC}(Y, Z)$. Since there are at most K entries in the UC matrix that need to be considered, finding the strongest replacement edge can be done in linear time.

Once the strongest replacement edge is found, then the *Sink-Dijkstra* procedure described earlier can be used to compute the updates to all of the $\mathcal{D}_x(X, Z)$ values. Fig. 6 summarizes the steps taken by the FAST-EX algorithm when a *contingent* time-point, C, happens to execute.

[12] Only the shortest edge between each pair of time-points is stored in the Ins and $Outs$ hash tables.

(1) For each core upper-case edge, $Y \xrightarrow{C:-w} A$, labeled by C, that edge is removed from the STNU graph (i.e., from the UC matrix) and the corresponding edge from Y to Z is removed from the *AllMax* graph. Then the strongest replacement edge from Y to Z in the *AllMax* graph is obtained from $\texttt{OC}(Y, Z)$ or one of at most K entries in the UC matrix.
(2) The edge, $Z - C \leq -t$ (i.e., $C \geq t$), is added to the *AllMax* graph (as is done when a non-contingent time-point executes).
(3) For each as-yet-unexecuted time-point, Y, the *greater-than-now* constraint, $Z - Y \leq -t$ (i.e., $Y \geq t$), is added to the *AllMax* graph (as is done when a non-contingent a time-point executes).
(4) The *Sink Dijkstra* procedure is used to compute all updates of the form, $\mathcal{D}_x(X, Z)$, for any X.
(5) The constraint, $C - Z \leq t$ (i.e., $C \leq t$), is added to the network (as is done when a non-contingent time-point executes).
(6) The *SourceDijkstra* procedure is used to compute all updates of the form, $\mathcal{D}_x(Z, X)$, for any X.

Fig. 6. FAST-EX's response to the execution of a contingent time-point, C

E. Putting it all Together

Given a dynamically controllable STNU, the FAST-EX algorithm initializes the following structures.

- Hash tables:
  - $\mathcal{U}_x$: the unexecuted executable time-points
  - $\mathcal{U}_c$: the unexecuted contingent time-points
  - $\mathcal{R}$: the rigid component, containing executed time-points, initially only Z
  - $Ins(X)$: for each X, the edges coming into X in $\mathcal{G}_x$
  - $Outs(X)$: for each X, the edges leaving X in $\mathcal{G}_x$
- The N-by-N *AllMax* distance matrix, $\mathcal{D}_x$, is that which exists at the end of Morris' DC-checking algorithm. Only the entries involving Z are used by FAST-EX.
- The *ordinary* core edges generated by Morris' algorithm are collected in an N-by-N matrix, called the *ordinary core* matrix, OC.
- The *upper-case* core edges generated by Morris' algorithm are collected in a K-by-N matrix, called the *upper-case core* matrix, UC.
- The variable, NOW, is initialized to 0.

Fig. 7 gives pseudo-code for one iteration of FAST-EX.

IF $\mathcal{U}_x$ and $\mathcal{U}_c$ are both empty, THEN done, ELSE:

1. Generate the next real-time execution decision, RD.
2. Observe the outcome of RD: $(\texttt{NOW}', \texttt{NewExec})$.
3. If no contingent time-points executed, then for each $X \in \texttt{NewExec}$:
   a. Process execution constraints, using *SourceDijkstra* and *SinkDijkstra*, as discussed in Section III-B (cf. Fig. 5).
   b. Move X from $\mathcal{U}_x$ to $\mathcal{R}$ and effectively remove X from the network by re-directing any edges involving X as discussed in Section III-C.
   c. Go to Step 6.
4. If only contingent time-points executed, then for each $C \in \texttt{NewExec}$:
   a. Process execution constraints and remove/replace edges corresponding to upper-case edges labeled by C, using *SinkDijkstra* and *SourceDijkstra*, as discussed in Section III-D (cf. Fig. 6).
   b. Move C from $\mathcal{U}_c$ to $\mathcal{R}$ and effectively remove C from the network by re-directing any edges involving C as discussed in Section III-C.
   c. Go to Step 6.
5. If both contingent and non-contingent time-points executed simultaneously, carry out Steps 3a-b for the non-contingent time-points and Steps 4a-b for the contingent time-points.
6. Go to the next iteration with $\texttt{NOW} := \texttt{NOW}'$.

Fig. 7. Pseudo-code for one iteration of the FAST-EX algorithm

F. Analysis of the FAST-EX Algorithm

This section presents results that confirm that the FAST-EX algorithm successfully executes any dynamically controllable STNU, and that it operates in $O(N^3)$ time overall: N time-point executions at $O(N^2)$-time per execution.

Theorem 2: Given any dynamically controllable STNU, $\mathcal{S}$, and the *AllMax* distance matrix computed by Morris' DC-checking algorithm, the FAST-EX algorithm guarantees the successful execution of $\mathcal{S}$. In particular, if the contingent durations fall within their specified bounds, then all constraints in the network will necessarily be satisfied.

Proof: This result derives from the fact that the FAST-EX algorithm makes the same execution decisions as the NEW-EX algorithm, which is proven by induction. Both algorithms start with the same *AllMax* distance matrix, $\mathcal{D}_x$, that results from Morris' DC-checking algorithm. Thus, both algorithms generate the same first decision. Given the same execution outcome, both algorithms also update all entries of the form, $\mathcal{D}_x(X, Z)$ and $\mathcal{D}_x(Z, X)$, which are the only entries used to generate subsequent execution decisions. (The NEW-EX algorithm also generates the rest of the $\mathcal{D}_x$ entries, but they are not at issue here.) The FAST-EX algorithm computes updates to the $\mathcal{D}_x$ entries by inserting execution constraints into the *AllMax* graph and propagating them using the *SinkDijkstra* and *SourceDijkstra* algorithms in alternation. When a contingent time-point, C, executes, the FAST-EX algorithm removes edges from the *AllMax* graph that correspond to upper-case edges labeled by C, and replaces them by the strongest corresponding edges arising from the corresponding OC matrix entry or upper-case edges labeled by other contingent time-points. Again, the *SinkDijkstra* and *SourceDijkstra* algorithms are used. The correctness of the updates depends on Theorem 1, which relies on the fact that the only constraints that are ever added to the network (or removed from it) involve Z. ■

Theorem 3: The worst-case performance of the FAST-EX algorithm is $O(N^3)$ time overall.

Proof: The worst-case performance of the FAST-EX algorithm is driven by the use of the *SinkDijkstra* and *SourceDijkstra* procedures, both of which run in $O(N^2)$ time. Execution of the STNU involves the execution of N time-points. When any time-point is executed, the *SinkDijkstra* and *SourceDijkstra* algorithms are run once each. Thus, the overall complexity is $O(N^3)$. Note that the cost of finding the replacement edge for each upper-case edge removed from the network is linear. Since at most NK edges are removed, that total cost over the entire run is also $O(N^3)$. ■

IV. Conclusions

This paper presented an $O(N^3)$-time incremental execution algorithm, called FAST-EX, that guarantees the successful execution of any STNU that has passed Morris' $O(N^4)$-time DC-checking algorithm. The FAST-EX algorithm makes the same execution decisions as the NEW-EX algorithm, but operates an order of magnitude faster: $O(N^3)$ time instead of $O(N^4)$ time. Like the NEW-EX algorithm, the FAST-EX algorithm's computations are spread out over the entire time the network is being executed; for FAST-EX: N iterations at $O(N^2)$ per iteration. Future work will empirically demonstrate the improved performance of the FAST-EX algorithm on a suite of randomly generated networks.

References

[1] R. Dechter, I. Meiri, and J. Pearl, "Temporal constraint networks," *Artificial Intelligence*, vol. 49, pp. 61–95, 1991.

[2] L. Hunsberger, "Magic loops in simple temporal networks with uncertainty," in *Proceedings of the Fifth International Conference on Agents and Artificial Intelligence (ICAART-2013)*, 2013.

[3] A. Gerevini, A. Perini, and F. Ricci, "Incremental algorithms for managing temporal constraints," IRST, Tech. Rep. IRST-9605-07, 1996.

[4] R. Wetprasit and A. Sattar, "Qualitative and quantitative temporal reasoning with points and durations (an extended abstract)," in *Fifth International Workshop on Temporal Representation and Reasoning (TIME-98)*, 1998, pp. 69–73.

[5] L. Hunsberger, "Group decision making and temporal reasoning," Ph.D. dissertation, Harvard University, 2002, available as Harvard Technical Report TR-05-02.

[6] P. Morris, N. Muscettola, and T. Vidal, "Dynamic control of plans with temporal uncertainty," in *17th International Joint Conference on Artificial Intelligence (IJCAI-01)*, B. Nebel, Ed. Morgan Kaufmann, 2001, pp. 494–499.

[7] L. Hunsberger, "Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies," in *Proceedings of the 16th International Symposium on Temporal Representation and Reasoning (TIME-2009)*. IEEE Computer Society, 2009, pp. 155–162.

[8] P. H. Morris and N. Muscettola, "Temporal dynamic controllability revisited," in *The Twentieth National Conference on Artificial Intelligence (AAAI-2005)*, M. M. Veloso and S. Kambhampati, Eds. The MIT Press, 2005, pp. 1193–1198.

[9] P. Morris, "A structural characterization of temporal dynamic controllability," in *Principles and Practice of Constraint Programming (CP 2006)*, ser. Lecture Notes in Computer Science. Springer, 2006, vol. 4204, pp. 375–389.

[10] L. Hunsberger, "A fast incremental algorithm for managing the execution of dynamically controllable temporal networks," in *Proceedings of the 17th International Symposium on Temporal Representation and Reasoning (TIME-2010)*. Los Alamitos, CA, USA: IEEE Computer Society, 2010, pp. 121–128.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.