Faster Dynamic Controllability Checking in Temporal Networks with Integer Bounds*

Nikhil Bhargava , Brian C. Williams
Massachusetts Institute of Technology
{nkb, williams}@mit.edu

Abstract

Simple Temporal Networks with Uncertainty (STNUs) provide a useful formalism with which to reason about events and the temporal constraints that apply to them. STNUs are in particular notable because they facilitate reasoning over stochastic, or uncontrollable, actions and their corresponding durations. To evaluate the feasibility of a set of constraints associated with an STNU, one checks the network's *dynamic controllability*, which determines whether an adaptive schedule can be constructed on-the-fly. Our work provides a dynamic controllability checker that is able to quickly refute the controllability of an STNU with integer bounds, such as those found in planning problems. Our work is faster than the existing best runtime for networks with integer bounds and executes in $O\left(\min(mn, m\sqrt{n}\log N) + km + k^2n + kn\log n\right)$. Our approach pre-processes the STNU using an existing $O(n^3)$ dynamic controllability checking algorithm and provides tighter bounds on its runtime. This makes our work easily adaptable to other algorithms that rely on checking variants of dynamic controllability.

1 Introduction

In temporal planning, an agent is presented with a series of events and must decide precisely when to schedule them. These events are often subject to constraints and the role of the agent is to construct a schedule that respects those constraints. This scheduling problem becomes more difficult when certain events are outside of their control. In such a situation, the agent is concerned with knowing whether a schedule can be constructed in real-time during execution, or whether the temporal problem is dynamically controllable.

In this paper, we introduce an improved algorithm for checking the dynamic controllability of Simple Temporal Networks with Uncertainty (STNUs) with integer bounds for its constraints. Our algorithm is based largely on two existing dynamic controllability algorithms: one that runs in $O(n^3)$ time [Morris, 2014] and a faster one that runs in $O(mn + k^2n + n\log n)$ [Cairo *et al.*, 2018], where n represents the number of events in a temporal network, m represents the number of temporal constraints, and k represents the number of uncontrollable events in the temporal network. Our algorithm applies some of the ideas from the faster algorithm, namely the use of a potential function to re-weight edges, to significantly speed up the $O(n^3)$ algorithm, dropping the overall runtime to $O\left(\min(mn, m\sqrt{n}\log N) + km + k^2n + kn\log n\right)$, where N is the magnitude of the most negative edge weight in the STNU's graphical representation. While our algorithm for checking controllability is incomplete, we provide a guarantee that it is correct whenever the algorithm marks a network as uncontrollable.

The requirement that all constraint bounds be integer is not overly restrictive in practice, as STNU bounds are often derived from human-specified characterizations and requirements on temporal events. When the integer constraint is relaxed, our algorithm still matches the best-known runtime of $O(mn + k^2n + n\log n)$. However, because we base our algorithm off of the $O(n^3)$ algorithm, which importantly uses length-preserving rules, we believe it is straightforward to transfer these same runtime improvements to other algorithms that rely on dynamic controllability checking variants, such as checking dynamic controllability for chain-free Partially Observable Simple Temporal Networks with Uncertainty [Bit-Monnot *et al.*, 2016] and checking the delay controllability of STNUs [Bhargava *et al.*, 2018], but this work is outside the scope of this current paper.

2 Background

Simple Temporal Networks (STNs) provide a way to formally model events, or timepoints, and the temporal constraints between them (e.g. event A must happen at least 20 minutes after event B) [Dechter *et al.*, 1991]. STNs, however, are incapable of modeling events that cannot be scheduled explicitly. These types of events are quite commonplace and effective agents must be capable of handling them in their plans (e.g. it is impossible to schedule that the time spent driving across town is exactly 22 minutes because it is highly

*This report is meant to amend an article by the same title that originally appeared in IJCAI-2019 [Bhargava and Williams, 2019]. The presentation of Theorem 3.1 in the original paper was incorrect, and this report amends the claims of the previous paper to correct the record.

dependent on traffic). Simple Temporal Networks with Uncertainty (STNUs) provide an explicit way to model this type of uncertainty [Vidal and Fargier, 1999].

Definition 1. STNU [Vidal and Fargier, 1999]
An STNU is a 4-tuple $\langle X_e, X_c, R_r, R_c \rangle$ where:

- X_e is the set of executable timepoints
- X_c is the set of contingent timepoints
- R_r is the set of requirement constraints of the form $l_r \leq x_r - y_r \leq u_r$, where $x_r, y_r \in X_b \cup X_e$
- R_c is the set of contingent constraints of the form $0 \leq l_r \leq c_r - e_r \leq u_r$, where $c_r \in X_c$, $e_r \in X_e$

STNUs subdivide their timepoints into a set of *executable timepoints* and *contingent timepoints*. Executable timepoints are those that are explicitly scheduled by the agent, and contingent timepoints represent stochastic events that are scheduled by nature. The constraints of an STNU are subdivided into *requirement constraints* and *contingent constraints*. Requirement constraints are ordinary constraints like those found in an STN. Contingent constraints impose a relation between a starting executable timepoint and an ending contingent timepoint describing when the contingent timepoint is guaranteed to happen in relation to the starting timepoint. When considering the runtime of algorithms over STNUs, we let m represent the total number of constraints, n the total number of timepoints, and k the total number of contingent timepoints.

Because many of the timepoints are outside of the control of the scheduler, it is often overly restrictive to construct a static schedule to determine the feasibility of an STNU. Instead, we often consider its *dynamic controllability* [Vidal and Fargier, 1999]. We say that an STNU is dynamically controllable if it is possible for an agent to construct a schedule during execution if they learn about the true value of contingent timepoints as they happen.

When trying to determine STNU dynamic controllability, we often prefer to work with their *labeled distance graph* representation [Morris, 2006]. In the labeled distance graph, each timepoint of the STNU corresponds to a node in the graph. Each requirement constraint of the form $u \leq B - A \leq v$ is split into two edges, $A \xrightarrow{v} B$ and $B \xrightarrow{-u} A$. Contingent constraints of the form $u \leq C - A \leq v$ (where C is a contingent timepoint) produce four edges. As was the case for requirement constraints, we produce edges $A \xrightarrow{v} C$ and $C \xrightarrow{-u} A$, but we also produce two *labeled* edges: the upper-case edge $C \xrightarrow{C:-v} A$ and the lower-case edge $A \xrightarrow{c:u} C$. It is clear by construction that there are $O(m)$ edges in the graph, $O(n)$ nodes in the graph, and $O(k)$ labeled edges.

These edges directly map to constraints that apply to the STNU that produced them. Unlabeled edges represent unconditional constraints that always apply whereas labeled edges represent conditional constraints that only apply if the label's corresponding contingent constraint takes on its longest or shortest possible value, for upper-case and lower-case labels respectively.

Just as constraints can be combined to create new constraints, so too can these edges be combined to produce new edges. [Morris, 2006] introduced a sound and complete set of edge reduction rules for use in an STNU's labeled distance graph:

- *Upper case reduction*: With edges $A \xrightarrow{x} B \xrightarrow{C:y} D$, produce edge $A \xrightarrow{C:(x+y)} D$.
- *Lower case reduction*: With edges $A \xrightarrow{b:x} B \xrightarrow{y} C$, if $y < 0$, produce edge $A \xrightarrow{x+y} C$.
- *Cross case reduction*: With edges $A \xrightarrow{b:x} B \xrightarrow{C:y} D$, if $y < 0, B \neq C$, produce $A \xrightarrow{C:x+y} D$.
- *No-case reduction*: With edges $A \xrightarrow{x} B \xrightarrow{y} C$, produce $A \xrightarrow{x+y} C$.
- *Label removal*: With edge $A \xrightarrow{B:x} C$, if $x \geq 0$, produce $A \xrightarrow{x} C$.

To determine whether an STNU is dynamically controllable, it suffices to determine whether the STNU's labeled distance graph has a semi-reducible negative cycle [Morris, 2006]. A semi-reducible negative cycle is a cycle whose total weight is negative and when after applying a series of reductions, we are left with a cycle without lower-case edges. The current best-known runtime for finding a semi-reducible negative cycle with this particular set of edge reductions is $O(n^3)$ [Morris, 2014].

In this paper we will use an augmented version of this ruleset and introduce the unconditional reductions, which comes in two flavors: unconditional-upper and unconditional-lower. While these rules are used to prove correctness, we do not use these rules directly in the execution of our algorithm. The following lemmas demonstrate that the two rules are sound.

Lemma 2.1. ***Unconditional-Upper Reduction***
If we start with edges $A \xrightarrow{C:u} D$ and $D \xrightarrow{v} B$, then we can add edge $A \xrightarrow{C:u+v} B$.

Lemma 2.2. ***Unconditional-Lower Reduction***
If we start with edges $A \xrightarrow{c:u} D$ and $D \xrightarrow{v} B$, then we can add edge $A \xrightarrow{c:u+v} B$.

Proof. We can use the same proof strategy for both lemmas. Let α be the label for the first edge in the sequence. The first edge provides the conditional constraint that $x_D - x_A \leq u$ that depends on the condition α (it holds either if C takes on its maximal or minimal value depending on if α is an upper or lower case label, respectively). The second edge provides the unconditional constraint that $x_B - x_D \leq v$. When we combine the two, we get that $x_B - x_A \leq u + v$ when the condition associated with α holds. This yields the output edge as specified in both lemmas.

□

While the current state-of-the-art algorithm uses a different set of rules for checking dynamic controllability [Cairo *et al.*, 2018], we show in this work that if we restrict our focus to constraints with integer bounds, we can continue to use the existing ruleset from [Morris, 2006] and improve on the best-known runtime for refuting dynamic controllability under any ruleset.

Input: A labeled distance graph $G = \langle V, E \rangle$ and non-negative potential function ψ
Output: Whether the STNU derived from the distance graph is free of semi-reducible negative cycles.
Initialization:
1 $negNodes \leftarrow$ the set of all vertices with incoming negative edges *under the potential function* ψ;
SRNCFREE?:
2 **for** $v \in negNodes$ **do**
3 $\quad cycleFree? \leftarrow$SRNCDIJKSTRA($G$, v, ψ, $[v]$, $negNodes$);
4 $\quad$ **if** $!cycleFree?$ **then**
5 $\quad\quad$ **return** $false$;
6 **return** $true$;

Algorithm 1: Semi-Reducible Negative Cycle Algorithm

3 Algorithm

Our contribution in this work is in demonstrating how a small set of modifications to the existing $O(n^3)$ algorithm for determining dynamic controllability [Morris, 2014] can lower the asymptotic worst-case runtime if we're interested primarily in refuting dynamic controllability. This improvement is primarily realized by modifying the underlying labeled distance graph of our input STNU by finding a graph re-weighting through the application of a potential function as used by [Cairo *et al.*, 2018].

We begin by introducing the core algorithms responsible for checking for the presence of a semi-reducible negative cycle and, by proxy, whether the STNU is dynamically controllable. Algorithms 1 and 2 are virtually identical to the ones presented in [Morris, 2014], and we rely quite heavily on the analysis in that work proving their correctness.

Briefly, the algorithm for checking dynamic controllability traverses the graph in reverse applying edge reductions as it goes in order to maintain a semi-reducible path. In its traversal, it uses a variant of Dijkstra's algorithm to ensure that it considers the shortest paths from its start node and has additional checks (e.g. line 18 of Algorithm 2) to guarantee that the path is semi-reducible. Dijkstra's algorithm, however, does not work in general with negative edges but will work properly if the only negative edge is the first one. The goal of the algorithm is to show that every walk from a negative edge eventually becomes positive; if so, there can be no semi-reducible negative cycles in the graph. If the dynamic controllability check runs into a negative edge that is not the first in its path during its walk, it recursively calls Dijkstra's algorithm again (line 14). If a sub-call returns, it adds all of its walks (which have positive length) to the graph so that callers can use them in their walks (line 10). If an infinite recursion is detected (line 5), then we know that each recursive call was made while it was in the midst of a semi-reducible negative walk, and the union of all of those walks yields a semi-reducible negative cycle. The $negNodes$ list in Algorithm 1 guarantees that all negative edges are considered.

This original variant of this algorithm runs in $O(n^3)$ time, and this can be seen by analyzing the runtime of SRNCFREE? (Algorithm 1) and the main algorithm it calls into, SRNCDIJKSTRA (Algorithm 2). For each call to SRNCDIJKSTRA and its terminal node, s, the algorithm runs a version of Dijkstra's algorithm over a subset of the graph that is composed of non-negative labeled and unlabeled edges, as well as positive edges that are derived through edge production rules. If no new edges were produced, each subcall would take $O(m + n \log n)$ time but instead we must argue that it takes $O(m' + n \log n)$ time where m' is the total number of edges in the graph after all edges are added into the graph. We know that each call to SRNCDIJKSTRA can add at most $O(n)$ new edges to the graph (by adding an edge from each node of the graph to s), so to bound the total number of new edges, we have to bound the number of calls to SRNCDIJKSTRA. In the worst case, there are n total calls to SRNCDIJKSTRA, meaning that the total runtime is $O\left(n(m + n^2 + n \log n)\right) = O(n^3)$ time.

The goal of our new set of algorithms is to minimize the number of calls to SRNCDIJKSTRA thus decreasing the overall runtime. The way we intend to do so is by introducing a potential function to re-weight the edge to decrease the total number of nodes with incoming negative edges.

3.1 Correctness under a Potential Function

We argue that by being smart about our choice of potential function, we can adapt an existing dynamic controllability checking algorithm and significantly improve its performance when checking for uncontrollability. We present such an adaptation below (Algorithms 1 & 2). The primary difference between the original controllability checking algorithm and the one presented here is that our modified function takes in a potential function ψ over the nodes that re-weights the edges of the graph such that the new weight $\tilde{w}$ of an edge going from u to v is given by $\tilde{w} = \psi(v) + w - \psi(u)$. The algorithm is evaluated with respect to the edge weights under the potential function and has a slight change (see line 18 of Algorithm 2) when checking whether lower- and cross-case edge reductions apply. With this change, our algorithm searches for a cycle that is negative under the potential function but semi-reducible when considering the original set of weights. Because a cycle's total weight is not affected by a potential function, the result is an algorithm that is able to find a semi-reducible negative cycle in our original graph.

To prove that our approach is appropriate when prioritizing correct evaluation of uncontrollability, we show that for certain potential functions, when we pass in an STNU that is controllable, our algorithm always correctly return true.

Though we know that applying a potential function preserves the overall length of cycles in the graph, it is not clear whether applying any potential function to an STNU's labeled distance graph guarantees that a semi-reducible negative cycle remains semi-reducible.

In particular, we care to show that after applying our potential function, a controllable network remains controllable and we do not erroneously reject controllable networks. While we may not be able to prove this reuslt in general, we can prove that this result holds for a specific subset of cases, namely for potential functions that are non-negative for all vertices. If we demonstrate this, then it is clear that if we find such a potential function and apply it to an STNU, we have a guarantee

Input: Labeled distance graph $G = \langle V, E \rangle$, terminal node s, non-negative potential function ψ, $callStack$, and negative nodes $negNodes$
Output: Whether the current walk is cycle-free
Initialization:
1 $Q \leftarrow PriorityQueue()$;
2 **for** $e \in s.incomingEdges()$ **do**
3 | **if** $e.weight(\psi) < 0$ *and* $!e.lowerCase()$ **then**
4 | | $Q.add(\langle e.from, e.label \rangle, e.weight(\psi))$;
SRNCDIJKSTRA:
5 **if** $s \in callStack[1 : end]$ **then**
6 | **return** $false$;
7 **while** $Q.size() > 0$ **do**
8 | $v, label, weight \leftarrow Q.pop()$;
9 | **if** $weight \geq 0$ **then**
10 | | $G.add(\langle v, s, weight \rangle)$;
11 | **else**
12 | | **if** $v \in negNodes$ **then**
13 | | | $newStack \leftarrow [v].concat(callStack)$;
14 | | | $result \leftarrow$ SRNCDIJKSTRA(G, v, ψ, $newStack$, $negNodes$);
15 | | | **if** $!result$ **then**
16 | | | | **return** $false$;
17 | | **for** $e \in v.incomingEdges()$ **do**
18 | | | **if** $e.weight(\psi) \geq 0$ *and* $!(e.isLowerCase()$ *and* $(e.label == label$ *or* $weight - \psi(s) + \psi(v) \geq 0))$ **then**
19 | | | | $w \leftarrow e.weight(\psi) + weight$;
20 | | | | $Q.addOrDecKey(\langle e.from, label \rangle, w)$
21 $negNodes.remove(s)$;
22 **return** $true$;

Algorithm 2: Function SRNCDIJKSTRA

that if the original STNU is dynamically controllable, our algorithm returns true. Note that for any potential function, it is trivial to create one that satisfies this condition; we simply add a large constant uniformly to all potential values, which leaves the induced edge weights entirely unaltered.

Theorem 3.1. *Given a dynamically controllable STNU S and a potential function ψ over S's labeled distance graph such that for each vertex v, $\psi(v) \geq 0$, Algorithm 1 returns true.*

Proof. To show that our algorithm will recognize S as dynamically controllable, we construct a new STNU S' that we use to analyze the algorithm's behavior. S' is identical to S except that every vertex v is split into v and v'. For every vertex v, a single requirement link is created $\psi(v) \leq v' - v \leq \psi(v)$. Because $\psi \geq 0$, it is clear that S' is also dynamically controllable because any valid strategy for S would similarly work for S' if it were augmented by a guarantee that for all v, v' was executed $\psi(v)$ units of time after v.

If we inspect the behavior of our algorithm on S and ψ, we observe that it simulates the behavior of the original $O(n^3)$ algorithm on S' with one minor deviation. As the algorithm walks the graph, every time it reaches some vertex v, it immediately takes the (reverse) $v' \xrightarrow{-\psi(v)} v$ edge. When that path is popped from the queue at a later point, its only option is to take the $v \xrightarrow{\psi(v)} v'$ edge in essence reversing its previous decision. Rather than exploring each detour one at a time, it groups them in three, as was described in the previous theorem. The modified check at line 18 again guarantees that lower-case edges are properly checked for elimination as their values are temporarily shifted because of the potential function. Thus, we know that if our algorithm were to return false, we would arrive at a contradiction because it would mean that S' is not dynamically controllable.

□

Input: A labeled distance graph $G = \langle V, E \rangle$ and potential function ψ
Output: Whether the STNU given by the labeled distance graph is dynamically controllable.
Initialization:
1 $loProj \leftarrow$ a projection of graph G that only contains lower-case and unlabeled edges with all labels removed;
2 $N \leftarrow$ the minimum value such that all edge weights satisfy $w \geq -N$ as well as $N \geq 2$;
CHECKDC:
3 **if** $(\sqrt{|V|} < \log N)$ *or G has non-integer constraints* **then**
4 | $\phi \leftarrow$ BELLMANFORDPOTENTIALS($loProj$);
5 **else**
6 | $\phi \leftarrow$ GOLDBERGPOTENTIALS($loProj$);
7 **if** $\phi == \emptyset$ **then**
8 | **return** $false$;
9 $\psi \leftarrow -\phi$;
10 $\psi.addToAll(-\psi.min)$;
11 **return** SRNCFREE?(G, ψ);

Algorithm 3: Full Dynamic Controllability Checking Algorithm

With this theorem, we have now shown that Algorithm 1 checks the dynamic controllability of an STNU and only returns false given a potential function if the original STNU is not controllable. However, we have neither shown how to derive this potential function nor why including a potential function would improve overall running time. In the rest of this section, we will explain these two points, ultimately demonstrating our desired runtime.

3.2 Generating Potential Functions & Runtime Complexity

In Algorithm 3, we introduce the full algorithm that we will use to evaluate the dynamic controllability of an STNU. Our algorithm starts by finding a potential function, making its values entirely non-negative, and then checking dynamic controllability with respect to that potential function. We will start by briefly describing the algorithms for deriving a potential function.

There are two functions that we use to derive our potential functions, BELLMANFORDPOTENTIALS (line 4, Algorithm 3) and GOLDBERGPOTENTIALS (line 6, Algorithm 3).

Input: A graph stripped of all labels $G = \langle V, E \rangle$
Output: A potential function ϕ or $\emptyset$ if G has a negative cycle
Initialization:
1 **for** $v \in V$ **do**
2 | $\phi(v) \leftarrow 0$;
BELLMANFORDPOTENTIALS:
3 **for** $i \in [1..(|V|-1)]$ **do**
4 | **for** $e \in E$ **do**
5 | | $\phi(e.start) \leftarrow \max(\phi(e.start), \phi(e.end) - e.weight)$;
6 **for** $e \in E$ **do**
7 | **if** $\phi(e.start) + e.weight - \phi(e.end) < 0$ **then**
8 | | **return** $\emptyset$;
9 **return** ϕ;

Algorithm 4: Algorithm for finding potential function, from [Cairo *et al.*, 2018]

Input: A graph stripped of all labels $G = \langle V, E \rangle$ and edge weights given by l
Output: A potential function ϕ or $\emptyset$ if G has a negative cycle
Initialization:
1 **for** $v \in V$ **do**
2 | $\phi(v) \leftarrow 0$;
3 $l_p \leftarrow l$;
4 $G^- \leftarrow G$ restricted to negative and zero-length edges;
GOLDBERGPOTENTIALS:
5 **if** $min(l) < -1$ **then**
6 | $l' \leftarrow \lceil \frac{l}{2} \rceil$;
7 | $\phi \leftarrow$ GOLDBERGPOTENTIALS(G, l');
8 | **if** $\phi == \emptyset$ **then**
9 | | **return** $\emptyset$;
10 | **for** $u, v \in V^2$ **do**
11 | | $l_p(u,v) \leftarrow l(u,v) + 2\phi(u) - 2\phi(v)$;
12 $components \leftarrow G^-.stronglyConnectedComps()$;
13 **for** $c \in components$ **do**
14 | **if** $c.hasNegativeEdge()$ **then**
15 | | **return** $\emptyset$;
16 | $G^-.contract(c)$;
17 $G^-.addNode(s)$;
18 **for** $v \in G^-.nodes()$ **do**
19 | $G^-.addEdgeWithWeight(s, v, 0)$;
20 **for** $i \in [1...|V|]$ **do**
21 | $L_i \leftarrow \{v \in G^-.nodes() | dist(s,v) = -i\}$;
22 $r \leftarrow max_r(r | L_r \neq \emptyset)$;
23 $q \leftarrow argmax_q(|L_q|)$;
24 **if** $r \geq \sqrt{k}$ **then**
25 | $\phi.adjustChain(L_r)$;
26 **else**
27 | $\phi.adjustLayer(L_q)$;
28 **return** ϕ;

Algorithm 5: Algorithm for finding potential function, from [Goldberg, 1995]. For more details and analysis, please see the original paper.

BELLMANFORDPOTENTIALS is elaborated in full in Algorithm 4 and maps exactly to a combination of INITPOTENTIAL and NEGATIVECYCLE from Algorithm 3 of [Cairo *et al.*, 2018]. It runs a variant of the Bellman-Ford algorithm computing a potential function along the way. If after a certain number of iterations, it has not found a potential function ϕ such that for all edges $u \xrightarrow{w} v$, $\phi(u) + w - \phi(v) \geq 0$, then it knows that the input must have a negative cycle.

GOLDBERGPOTENTIALS (Algorithm 5) is reproduced from the pseudocode in [Goldberg, 1995]. The algorithm attempts to construct a potential function ϕ that for each edge $u \xrightarrow{w} v$, $\phi(u) + w - \phi(v) \geq 0$; if it is unable to do so, it determines that there must be a negative cycle in the graph. The algorithm builds its potential function using a scaling mechanism and at each level either finding a long chain of potential values it can adjust at once or a large layer of potential values to adjust at once. Because it successively scales its arguments, it requires that all edge weights be integer. Overall, the algorithm runs in $O(m\sqrt{n}\log N)$ time.

By default, these algorithms do not produce the potential function we want. Our proofs depended on having a potential function that satisfied $\psi(v) + w - \psi(u) \geq 0$. If we let $\psi = -\phi$ (line 9), then the two sub-calls would yield $-\psi(u) + w + \psi(v) \geq 0$, as we wanted. This potential function, however, may not be positive, but if our algorithm is considering a potential function, line 10 of Algorithm 3 will ensure that the potential function has no non-negative values. What remains is to consider what happens when we fail to find a potential function.

Both algorithms will fail to yield a potential function if and only if the input $loProj$ has a negative cycle, and when we cannot find a potential function, Algorithm 3 returns false. If we assumed that this response were incorrect, i.e. that the input $loProj$ had a negative cycle but the STNU were dynamically controllable, we would quickly run into a contradiction. If $loProj$ had a negative cycle, then that means that there exists some combination of unlabeled and lower-case edges that form a cycle with negative weight. However, if we considered the case where all of our contingent constraints took on their minimum possible value, we would enforce the constraints of all lower-case edges since their conditions apply. In such an instance, we would be unable to execute our STNU. To see this, consider the constraints that we would derive by combining the constraints associated with adjacent edges. Each pair of adjacent edges are represented by constraints of the form $C - B \leq u$ and $B - A \leq v$ and yield constraints of the form $C - A \leq u + v$. Because the edges form a cycle, we would eventually reach a point where we would have two constraints $C - A \leq w$ and $A - C \leq w'$ that would combine to form $0 \leq w + w'$. But because the edge production rules are length-preserving, we know that $w + w'$ represents the total weight of the cycle which is negative, meaning we have a contradiction. Thus, our algorithm will correctly determine an STNU's dynamic controllability, and we can now provide a thorough analysis of our algorithm's runtime.

Theorem 3.2. *Algorithm 3 runs in worst-case* $O\left(\min(mn, m\sqrt{n}\log N) + km + k^2 n + kn\log n\right)$ *time for STNUs with integer bounds.*

Proof. First, we look at the runtimes of the lines used to derive the potential functions (lines 4, 6). BELLMANFORDPOTENTIALS runs a variant of the Bellman-Ford algorithm to search for a negative cycle and constructs a potential function if one does not exist. It iterates through each edge for as many times as there are nodes in the graph, causing it to run in $O(mn)$ time. In contrast, GOLDBERGPOTENTIALS uses a scaling algorithm to either find a potential function or determine that the graph has a negative cycle; this algorithm takes $O(m\sqrt{n}\log N)$ time and requires that all edge weights be integers [Goldberg, 1995]. We know that if $\sqrt{n} < \log N$, then, in the limit, BELLMANFORDPOTENTIALS runs faster than GOLDBERGPOTENTIALS. Because we selectively choose which function to run based on exactly that inequality (line 3), we know that the set of operations through line 8 takes $O\left(\min(mn, m\sqrt{n}\log N)\right)$ time. If the search for a potential function instead yields a negative cycle, then we terminate immediately, satisfying the worst-case runtime.

We now consider what happens if we our initial checks yield a valid potential function. The update at line 9 ensures that the minimum value of ψ is 0 and involves at most $O(n)$ time to search for the minimum value and then update each value of ψ. This does not change the re-weighted values of any edges under ψ as the start and endpoints of each edge change by the same amount.

What remains is to analyze the runtime of SRNCFREE? (Algorithm 1). We begin by looking at SRNCDIJKSTRA (Algorithm 2). By the original analysis, if we let c be the number of times that SRNCDIJKSTRA is called, the overall runtime of the algorithm is given by $O\left(c(m + cn + n\log n)\right)$. The worst case is that all nodes have incoming negative edges and so SRNCDIJKSTRA is called once for every node. However, in our work we invoke the algorithm once for every node with an incoming negative edge *under the potential function* ψ. Our construction of ψ guarantees that when it is applied to all lower-case and unlabeled edges, the new edges are all non-negative. Thus, the only edges that can be negative under ψ are upper-case edges. Because there is exactly one upper-case edge per contingent link of STNU, SRNCDIJKSTRA is called at most $O(k)$ times and at most $O(kn)$ edges are added to the graph in total.

Thus, SRNCFREE? takes $O\left(k(m + kn + n\log n)\right)$ time, which means that our total runtime is $O\left(\min(mn, m\sqrt{n}\log N) + km + k^2n + kn\log n\right)$.

□

If we do not have a guarantee that our temporal constraint bounds are all integer, much of the same analysis can be applied, but we instead always use BELLMANFORDPOTENTIALS. This brings the total runtime to $O(mn + k^2n + kn\log n)$, which matches the best runtime for general dynamic controllability checking [Cairo *et al.*, 2018]. Unlike the other algorithm, however, our approach does not require any updates to the potential function as our algorithm is able to use the existing ruleset from [Morris, 2006] which guarantees that all transformations are length-preserving. This approach means that other algorithms that are heavily based off of the [Morris, 2014] algorithm, such as checking the dynamic controllability of chain-free POSTNUs [Bit-Monnot *et al.*, 2016] and checking the delay controllability of STNUs [Bhargava *et al.*, 2018], can likely similarly improve their runtimes to the stated bounds with minimal changes to their core algorithms.

4 Conclusion

In this paper, we introduce a modification to an existing $O(n^3)$ algorithm for determining dynamic controllability and show how this slight change speeds up the theoretical worst-case result if we are interested in quickly eliminating temporal networks that are known to be uncontrollable. Our final algorithm runs in $O\left(\min(mn, m\sqrt{n}\log N) + km + k^2n + n\log n\right)$ for temporal networks with integer bounds, which is an improvement on the best sound and complete dynamic controllability checking algorithm. Even when relaxing the integer bound requirement, our algorithm matches the best available runtime for checking controllability but the simplicity of the change means that this algorithm strategy is likely to apply to many other variants of STNU dynamic controllability checking.

We believe that this approach also extends quite naturally and can be used to build a sound and complete dynamic controllability checker with the same overall runtime. The approach from [Cairo *et al.*, 2017] uses a potential function but only uses Bellman-Ford to derive and update it. By using Goldberg's algorithm, it may be possible to improve on that runtime for networks with integer bounds.

References

[Bhargava *et al.*, 2018] Nikhil Bhargava, Christian Muise, Tiago Vaquero, and Brian Williams. Delay controllability: Multi-agent coordination under communication delay. In *DSpace@MIT*, 2018.

[Bit-Monnot *et al.*, 2016] Arthur Bit-Monnot, Malik Ghallab, and Félix Ingrand. Which contingent events to observe for the dynamic controllability of a plan. In *International Joint Conference on Artificial Intelligence (IJCAI-16)*, 2016.

[Cairo *et al.*, 2017] Massimo Cairo, Carlo Combi, Carlo Comin, Luke Hunsberger, Roberto Posenato, Romeo Rizzi, and Matteo Zavatteri. Incorporating decision nodes into conditional simple temporal networks. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 90. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[Cairo *et al.*, 2018] Massimo Cairo, Luke Hunsberger, and Romeo Rizzi. Faster dynamic controllability checking for simple temporal networks with uncertainty. In *The 25th International Symposium on Temporal Representation and Reasoning (TIME)*, 2018.

[Dechter *et al.*, 1991] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

[Goldberg, 1995] Andrew V Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[Morris, 2006] Paul Morris. A structural characterization of temporal dynamic controllability. In *International Conference on Principles and Practice of Constraint Programming*, pages 375–389. Springer, 2006.

[Morris, 2014] Paul Morris. Dynamic controllability and dispatchability relationships. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 464–479. Springer, 2014.

[Vidal and Fargier, 1999] Thierry Vidal and Helene Fargier. Handling contingency in temporal constraint networks: from consistency to controllabilities. *Journal of Experimental & Theoretical Artificial Intelligence*, 11(1):23–45, 1999.