Bucket Elimination Algorithm for Dynamic Controllability Checking of Simple Temporal Networks with Uncertainty

Yuening Zhang, Brian Williams
Massachusetts Institute of Technology
77 Massachusetts Ave
Cambridge, MA 02139

Abstract

Simple Temporal Networks with Uncertainty (STNU) can represent temporal problems where duration between events may be uncontrollable, e.g. when the event is caused by nature. An STNU is dynamically controllable (DC) if it can be successfully scheduled online. In this paper, we introduce a novel usage of bucket elimination algorithms for DC checking that matches the state of the art in achieving $O(n^3)$ performance. Bucket elimination algorithms exist for STNs (path consistency and Fourier algorithms), but adapting it to STNUs is non-trivial. As a result, consistency checking becomes a special case of our algorithm. Due to the familiarity to bucket elimination algorithms, the final algorithm is easier to understand and implement. Additionally, conflict extraction is also easily supported in this framework.

Introduction

Temporal networks are a representation widely adopted for modeling scheduling problems, where binary inequality constraints are placed on pairs of events. While simple temporal networks (STN) [Dechter, Meiri, and Pearl1991] assume full control over execution of all events, simple temporal networks with uncertainty (STNU) (Definition 1) allow specifying *received events* that can only be observed by the agents. Each received event requires specification of the duration bound from its unique *activated event*, which is called a *contingent constraint* which we differentiate from the original *requirement constraints*. [Vidal1999] proposes three notions of controllability for STNUs, similar to consistency for STNs, which are weak controllability (WC), strong controllability (SC) and dynamic controllability (DC). Among these, DC is the most interesting and commonly used one as being dynamically controllable means that an execution policy exists that can dispatch the events based on previous observations.

Definition 1 (STNU [Vidal1999])**.** An *STNU* is a tuple $\langle X_e, X_c, R_r, R_c \rangle$, where

- X_e is a set of executable events.
- X_c is a set of received events.
- R_r is a set of requirement constraints of the form $lb \leq x_j - x_i \leq ub$, where $x_i, x_j \in X_e \cup X_c$.
- R_c is a set of contingent constraints of the form $lb \leq x_j - x_i \leq ub$, where $x_i \in X_e, x_j \cup X_c$, and x_i is the activated event for x_j.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

While classic bucket elimination algorithms exist for checking consistency of STNs, such as Fourier algorithms [Dechter1999] and path consistency [Boerkoel and Durfee2013], extending them to dynamic controllability checking is non-trivial. On the other hand, efficient polynomial algorithms exist for dynamic controllability checking as well as its incremental version [Morris2014, Nilsson, Kvarnström, and Doherty2016, Bhargava, Vaquero, and Williams2017, Cairo, Hunsberger, and Rizzi2018, Bhargava and Williams2019].

In this paper, we introduce a novel bucket elimination algorithm for checking dynamic controllability that matches the state of the art in $O(n^3)$ performance. The bucket elimination framework is interesting as it presents familiarity to people in constraint programming and AI community, and does not require any special algorithmic design such as recursive reverse Dijkstra's algorithm [Morris2014], thus is potentially easier to understand and implement. It also presents benefits such as easy support for conflict extraction, and has the potential to apply techniques in the bucket elimination literature such as variable ordering heuristics, parallelization etc.

Background

A simple temporal constraint with start event s, end event e, lower bound lb and upper bound ub can be equivalently represented as a conjunction of two upper bound inequalities $(s - e \leq -lb) \wedge (e - s \leq ub)$. They can thus be represented by directed edges in a *distance graph* (DG), and consistency checking is determined by finding negative cycles in the graph [Dechter, Meiri, and Pearl1991]. An example is shown in Figure 1.

For dynamic controllability checking, Morris [Morris2006] showed that an STNU can be represented by a *labeled distance graph* (LDG), where edges of a contingent constraint are labelled with upper or lower cases of the observed event [Morris2006], as shown in Figure 2. Dynamic

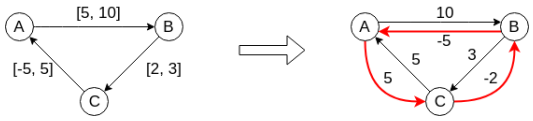

Figure 1: Left: STN, Right: DG. The STN is inconsistent, evidenced by the negative cycle highlighted in red

controllability is verified based on the existence of semi-reducible negative cycles (Definition 2), i.e. negative cycles containing no lowercase edges after applying a series of reduction rules (Figure 6) proposed in [Morris2006]. Figure 3 shows an example of uncontrollable STNU. Our bucket elimination algorithm for checking dynamic controllability draws insights from Morris's work [Morris2006, Morris2014], and uses the same principle to check dynamic controllability (Theorem 1).

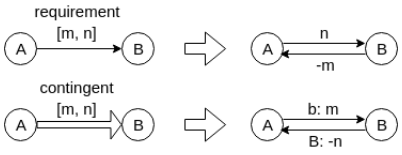

Figure 2: Conversion of temporal constraints to labeled edges in a labeled distance graph

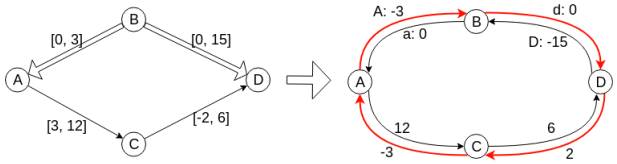

Figure 3: Let: STNU (double arrows are contingent constraints), Right: LDG. The STNU is not dynamically controllable, as evidenced by the semi-reducible negative cycle highlighted in red.

Definition 2 (Semi-Reducible Path)**.** A path is reducible if it can be transformed into a single edge by a sequence of reductions. A path is semi-reducible if it can be transformed into a path without lower-case edges by a sequence of reductions.

Theorem 1. (Dynamic Controllability) An STNU is dynamically controllable if and only if it does not have a semi-reducible negative cycle.

Our work leverages bucket elimination algorithms, which are a family of inference algorithms characterized by sequential processing of buckets [Dechter1999]. Bucket elimination has been used to solve constraint satisfaction problems (CSP). Whereas consistency checking can be cast as a CSP, dynamic controllability checking cannot, as it is a temporal problem where the values of the observed events may only be determined as they occur. Bucket elimination algorithms have been studied for consistency checking, including path consistency [Boerkoel and Durfee2013] and Fourier algorithms [Dechter1999], but are limited to STNs. Our work extends the bucket elimination framework to dynamic controllability checking.

Algorithm

The high-level skeleton of the bucket elimination algorithm is illustrated in Algorithm 1. The algorithm takes a standard approach of bucket elimination algorithms — it carries out inference through sequential elimination of variables as buckets. The algorithm takes the converted input of a *labeled distance graph* from STNU, and outputs whether the STNU is dynamically controllable, the conflict, if any, and the elimination order. If the STNU is not dynamically controllable, the returned conflict is a negative cycle in the original uneliminated LDG that proves why the network is uncontrollable. As shown in Figure 4, nodes D and C are eliminated in order, after which a semi-reducible negative cycle between nodes A and B is identified to prove that the network is not controllable.

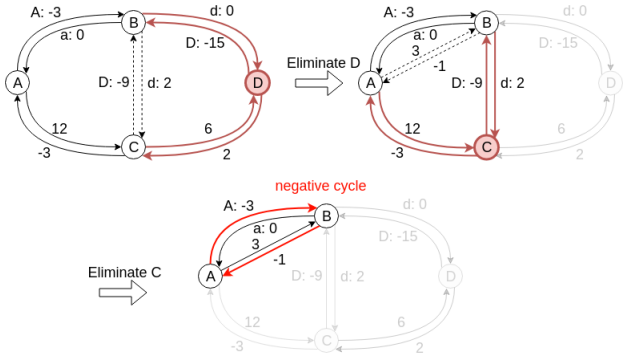

Figure 4: The algorithm eliminates one node at the time, triangulating pair-wise edges at each step of the elimination and checking for semi-reducible negative cycles, until a semi-reducible negative cycle is found (network is not controllable) or all nodes are eliminated (controllable).

Algorithm 1: BucketElimination

Input : LDG $G = \langle V, E \rangle$
Output: Boolean $feasible$, conflict cf, elimination order o

$o \leftarrow \emptyset$
$G' \leftarrow G$
while $v, nc \leftarrow$ NEXTNODE(G') **do**
  if $v \neq None$ **then**
    $feasible, nc, G' \leftarrow$ ELIMINATE(G', v)
    if $\neg feasible$ **then**
      return $False,$ EXTRACTCONFLICT$(nc), o$
    $o \leftarrow o \cup v$
  else if $nc \neq None$ **then**
    return $False,$ EXTRACTCONFLICT$(nc), o$
return $True, None, o$

The main bucket elimination loop (line 3 - 10) iteratively finds the next node to eliminate. If the network is found to be uncontrollable (line 6, 9), the algorithm terminates and extracts the conflict (line 7, 10).

The sub-function NEXTNODE (line 3) takes the current graph G' and looks for the next node v to be eliminated. If a node v is found, NEXTNODE returns the nodes v for elimination, as is the case for nodes D and C. If not, then NEXTNODE checks for any negative cycles consisting only of negative-weight edges. If there is, the network is not controllable, and the negative cycle nc is returned, as is the case for the negative cycle between nodes A and B. Otherwise, it means all the nodes have been eliminated, and the network is controllable. In the case of consistency checking for STNs, it simply applies heuristics such as Min-Degree [Bodlaender and Koster2010] to reduce the induced tree width. For dynamic controllability checking, the elimination order is also constrained by a specific partial order, which we will introduce in the next section.

Algorithm 2: Eliminate

Input : LDG $G = \langle V, E \rangle$, v
Output: Boolean $feasible$, negative cycle nc, LDG G'

1 $E_{elim} \leftarrow$ GETBUCKET(v)
2 $feasible, nc, E_{tri} \leftarrow$ JOINPROJECT(v, E_{elim})
3 **if** $\neg feasible$ **then**
4 **return** $False, nc, G$
5 $G' \leftarrow$ ADDTIGHTEST(G, E_{tri})
6 $G' \leftarrow$ REMOVE(G', E_{elim})
7 $G' \leftarrow$ REMOVE(G', v)
8 **return** $True, None, G'$

The sub-function ELIMINATE is shown in Algorithm 2. Each node is a bucket that contains all directed edges with the node being either the source or the target (line 1). For example, in Figure 4, the node to be eliminated, such as D, and its bucket of edges is highlighted in dark red. When a node is eliminated, all the edges are joined and projected onto its connecting nodes (line 2), as shown by the dashed edges in Figure 4. Since our temporal constraints are binary constraints represented by directed edges, the operation involves triangulation of every pair of directed edges. The triangulated edges, if *tightest*, i.e. intuitively, no shorter or equal distance edge with the same label exist between the nodes (fully described in next section), are added to the graph (line 5). For example, in Figure 4, the edge from B to A labeled with -1 is *tighter* than the one labeled with $a : 0$; hence the latter can be removed without affecting the dynamic controllability of the network. Finally, the node as well as its bucket are eliminated from the graph (line 6-7).

The sub-function JOINPROJECT is shown in Algorithm 3. It goes through each pair of edges that points into node v and points out of node v, and triangulates them to produce a new edge (line 9), as illustrated in Figure 4. If the edges form a cycle, then instead of triangulating them, it checks whether the cycle is a semi-reducible negative cycle (line 5). If it is, the edges are returned as the negative cycle nc. The

Algorithm 3: JoinProject

Input : v, E_{elim}
Output: Boolean $feasible$, negative cycle nc, E_{tri}

1 $E_{tri} \leftarrow \{\}$
2 **for** e_i in $E_{elim}, source(e_i) == v$ **do**
3 **for** e_j in $E_{elim}, target(e_j) == v$ **do**
4 **if** $target(e_i) == source(e_j)$ **then**
5 $feasible \leftarrow$ CHECKNEGCYCLE(e_i, e_j)
6 **if** $\neg feasible$ **then**
7 **return** $False, \{e_i, e_j\}, \{\}$
8 **else**
9 $E_{tri} \leftarrow E_{tri} \cup$ TRIANGULATE(e_i, e_j)
10 **return** $True, None, E_{tri}$

rules under which CHECKNEGCYCLE and TRIANGULATE triangulates edges and checks controllability are introduced next.

Dynamic Controllability Checking

We elaborate on the functions TRIANGULATE, CHECKNEGCYCLE, ADDTIGHTEST, and NEXTNODE, which involve non-trivial modifications to check dynamic controllability.

Triangulation and Controllability Checking TRIANGULATE and CHECKNEGCYCLE take the input of a pair of edges, and apply a triangulation operation to produce a triangulated edge or check whether the edges form a semi-reducible negative cycle. In the degenerate case of checking consistency, the triangulation rule simply follows from the triangle inequality, as shown in Figure 5. CHECKNEGCYCLE additionally checks consistency based on evidence of a negative cycle, that is, if $x + y < 0$ in Figure 5.

$$A \xleftarrow{x} C \xleftarrow{y} D \quad \text{adds} \quad A \xleftarrow{x+y} D$$

Figure 5: STN triangulation rules

To check dynamic controllability, we need a different set of triangulation rules that take care of labeled edges. We adopt the set of symmetric reduction rules from [Morris2006], summarized in Figure 6, as well as the additional rules in Figure 7. Example triangulations that apply these rules can be seen in Figure 4. Similarly, CHECKNEGCYCLE checks dynamic controllability by looking for evidence of a negative cycle after triangulation, that is, if the sum of the edges applicable in the triangulation is negative. Adopting the symmetric reduction rules also requires the STNU to be in *normal form*. An STNU is in *normal form* if the lower bound of every contingent constraint is 0. Any STNU can be converted into an equivalent normal form [Morris2006] by replacing each contingent constraint of $[lb, ub]$ with a requirement constraint of $[lb, lb]$, and a contingent constraint of $[0, ub - lb]$.

While the original set of rules from [Morris2006] (Figure 6) is sound, we adopt additional rules (Figure 7) because we eliminate the parent edges after processing each node, and

(Upper-Case Reduction)
$A \xleftarrow{B:x} C \xleftarrow{y} D$ adds $A \xleftarrow{B:(x+y)} D$

(Lower-Case Composition)
$A \xleftarrow{x} E \xleftarrow{c:y} D$ adds $A \xleftarrow{c:(x+y)} D$

(Cross-Case Composition) If $B \neq C$,
$A \xleftarrow{B:x} E \xleftarrow{c:y} D$ adds $\begin{cases} A \xleftarrow{B:(x+y)} D \text{ if } (x+y) < 0 \\ A \xleftarrow{c:(x+y)} D \text{ if } (x+y) \geq 0 \end{cases}$

(Lower Label Removal) If $z < 0$,
$A \xleftarrow{c:z} D$ adds $A \xleftarrow{z} D$

(Label Removal) If $z \geq 0$,
$A \xleftarrow{B:z} C$ adds $A \xleftarrow{z} C$

(No-Case Reduction)
$A \xleftarrow{x} C \xleftarrow{y} D$ adds $A \xleftarrow{x+y} D$

Figure 6: Triangulation rules carried over from the symmetric reduction rules in [Morris2006]

(Lower Label Overapproximation)
$A \xleftarrow{b:x} E \xleftarrow{y} D$ adds $A \xleftarrow{(x+y)} D$
$A \xleftarrow{b:x} E \xleftarrow{c:y} D$ adds $A \xleftarrow{c:(x+y)} D$

Figure 7: Additional triangulation rules

rules in [Morris2006] alone do not guarantee completeness for our bucket elimination algorithm. Eliminated edges can cause loss of information and the algorithm may fail to discover semi-reducible negative cycles. An example is shown in Figure 8. The lower label over-approximation rule in Figure 7, together with the canonical partial elimination order described later in this section, ensures that the transformation of the graph before and after each elimination preserves dynamic controllability.

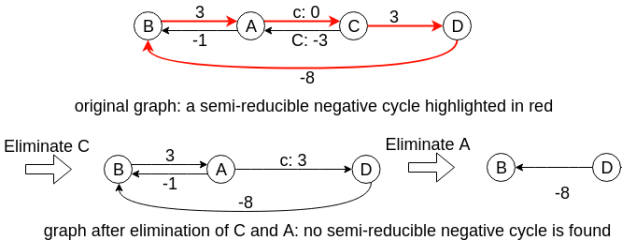

Figure 8: The initial labeled distance graph on the top has a semi-reducible negative cycle highlighted in red. If we eliminate C and A, then notice that when eliminating A, the rules in Figure 6 do not specify how to triangulate e_{BA} and e_{AD}. Therefore, without the lower-label approximation rule, the algorithm would fail to detect the negative cycle.

Tightest Edges The triangulation operations create more edges along the way and the edges may compound, resulting in elimination being very slow. Notice that not all the edges need to be triangulated, as some of them are dominated by others. We reduce the number of edges that need to be considered in triangulation by defining the *tightest edges*. In the degenerate case of checking STN consistency, we only need to keep the shortest edge between two nodes. In the same way, for checking dynamic controllability, we only need to keep the edges that are *tightest*, defined in 3.

Definition 3 (Tightest Edge)**.** For two edges e_i and e_j that have the same source node and target node, edge e_i is *tighter* than edge e_j if and only if by removing e_j and keeping e_i, the dynamic controllability of the network remains the same. Edge e_i is *strictly tighter* than e_j if and only if e_i is tighter than e_j, but e_j is not tighter than e_i. A *tightest* edge is an edge where no other edges are strictly tighter than it.

For labeled edges, we summarize how edge e_i can be tighter than e_j in Figure 9. In short, for e_i to be tighter than e_j, it has to satisfy two conditions: (1) e_i's weight is less than or equal to e_j's weight, (2) either e_i has no label, or e_i and e_j has the same label. If two or more edges are mutually tighter than each other, we only need to keep one of them.

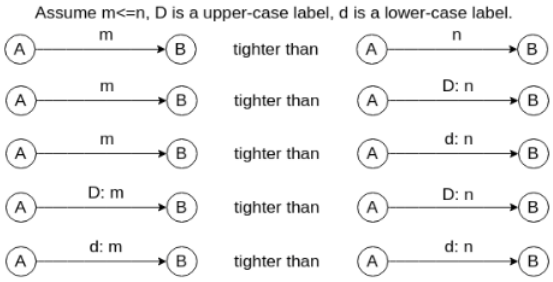

Figure 9: Tighter relationship

Next Node: Canonical Partial Elimination Order Bucket elimination algorithm allows processing of each node and its neighboring edges only once. This restricts us from applying the reduction rules repeatedly on an eliminated edge. However, not all combination of edges are covered by the set of triangulation rules (e.g. a upper-case edge followed by a lower-case edge), and eliminating the nodes in arbitrary order may cause the algorithm to be incomplete, as shown in Figure 10. The state-of-the-art DC checking algorithm in [Morris2014] walks along the edges in the reverse direction and recursively calls Dijkstra's algorithm whenever an unseen negative edge is encountered. The recursive call returns when the negative edge has an extension path that renders the path non-negative. Inspired by their algorithm, our algorithm eliminates the nodes in the order from their deepest recursive calls to the shallowest, so that the non-negative paths from deeper recursive calls can be used by shallower calls. As it turns out, this partial order, defined in Definition 4, exactly reflects the reverse-chronological order of the events happening, the same insight illustrated by some incremental dynamic controllability checking algorithms [Shah et al.2007, Nilsson, Kvarnström, and Doherty2016]. An example elimination that il-

lustrates the canonical partial order constraints and the corresponding order of elimination is shown in Figure 11.

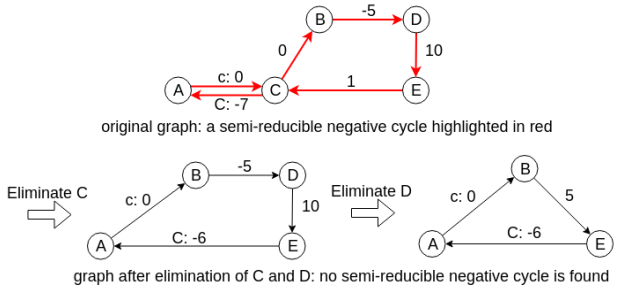

Figure 10: An example where eliminating node D before B causes the algorithm to be incomplete

Definition 4. (Canonical Partial Elimination Order) An elimination order o is said to follow the *canonical partial elimination order* if and only if

- source(e) $\prec_o$ target(e), for all $e \in \overline{E}$ where $w(e) < 0$.

where $\overline{E}$ denotes all the edges in the labeled distance graph throughout the bucket elimination process, including original edges and the triangulated edges, $a \prec_o b$ denotes a precedes b in the elimination order o and $w(e)$ denotes the weight of the directed edge e.

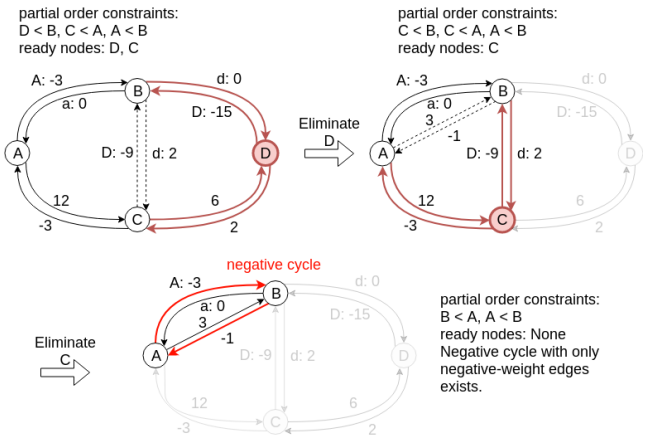

Figure 11: Initially, we have $D \prec_o B$, $C \prec_o A$, $A \prec_o B$, with D, C being ready. We randomly break tie and pick D to eliminate. This introduce edges in between B and C, and $C \prec_o B$. C is then eliminated, at which point there is no more ready node, but a semi-reducible negative cycle between A and B is detected, shown in red.

Based on the canonical partial order, we define *ready nodes* in Definition 5, that are nodes ready to be eliminated. NEXTNODE in the case of dynamic controllability checking finds the next node that is ready, while using heuristics to break ties to reduce induced tree width. The example in Figure 11 shows the ready nodes at each step given the canonical partial order constraints. When NEXTNODE fails to find the next node, Theorem 2 shows that it either determines the network is dynamically controllable or the opposite. If the network is not controllable, NEXTNODE returns a negative cycle as the conflict. As a result of the canonical partial elimination order, other combinations of edges for which our set of triangulation rules do not cover will not appear in the elimination process.

Definition 5 (Ready Nodes)**.** A node is *ready* if and only if it has no negative incoming edges, disregarding the labels.

Theorem 2 (Termination Condition)**.** If there remain nodes in the graph, but no ready node is found, then there are two cases: 1) either no node has negative outgoing edges, in which case the network is dynamically controllable, 2) otherwise, the network is not dynamic controllable and the remaining graph has at least one cycle with only negative-weight edges.

Proof. The first case is simple to prove. If there is no negative edge present in the graph, then there cannot be any negative cycles. To prove the second case, notice that if the graph has a cycle with only negative-weight edges such as in Figure 12, then the network is not dynamic controllable, because only unlabeled edges and upper-case edges can have negative weight, meaning the negative cycle is a semi-reducible negative cycle.

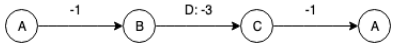

Figure 12: A cycle with only negative-weight edges.

We now prove that if there is no ready node, but some node exists that has negative outgoing edges, then the remaining graph has at least one cycle with only negative-weight edges. Suppose that in the graph there are n nodes V_{negout} with negative outgoing edges, and $m + n$ nodes V_{negin} with negative incoming edges, where $n > 0, m \geq 0$. We know that $V_{negout} \subseteq V_{negin}$ (otherwise a ready node could have been found), and each node $v \in V_{negout}$ has at least a negative incoming edge from V_{negout}. Assume by contradiction that there is no cycle with only negative-weight edges formed by V_{negout}, then suppose we have a sub-path AC (1) as illustrated in Figure 13. Since A has an incoming negative edge, and it cannot come from any node in AC, it must come from a node in another sub-path (2). By induction, we exhaust all of the n nodes in V_{negout} at sub-path (k), and the start of sub-path (k) must have an incoming edge from some node in all the aforementioned sub-paths. Hence a cycle with only negative-weight edges is formed, and the network is not dynamic controllable.

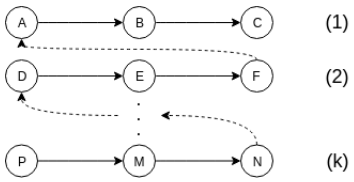

Figure 13: Sub-paths will eventually form a cycle with only negative-weight edges.

□

According to Theorem 2, the termination condition is met when NEXTNODE finds no ready node. NEXTNODE then does a final check for dynamic controllability by looking for nodes with negative outgoing edges that form a cycle, and returns the cycle as a conflict if the network is not controllable.

Correctness and Complexity Analysis

The STNU triangulation rules are sound and complete, that is, the network preserves its dynamic controllability before and after each elimination. As a result, the bucket elimination algorithm for checking dynamic controllability is sound and complete. The complexity of the algorithm is $O(n^3)$, which matches the state-of-the-art DC checking algorithm by Morris in [Morris2014]. The full proof for the theorems presented in this section is given in the Supplementary Materials.

Theorem 3. The triangulation rules are sound and complete in the bucket elimination algorithm for checking dynamic controllability.

Theorem 4. The bucket elimination algorithm for dynamic controllability checking is sound and complete.

Theorem 5. The complexity of the bucket elimination algorithm for checking dynamic controllability is $O(n^3)$, where n is the number of nodes.

Proof. We sketch the proof here, while readers can see Supplementary Materials for the full proof. The run time of the algorithm is dominated by the total number of triangulations done. Since only the tightest edges participate in triangulation, we only consider tightest edges between two nodes.

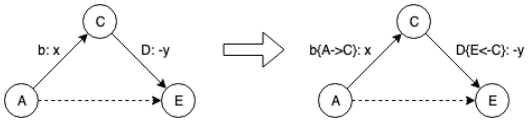

Figure 14: Relabelling the labeled edges.

To show that the number of triangulations is $O(n^3)$, we relabel the edges in the following way (Figure 14): for a lowercase edge from A to C with label b, we relabel the edge with $b_{A\to C}$, and for an uppercase edge from C to E with label D, we relabel the edge with $D_{E\leftarrow C}$. We further denote $\sigma_{A\to C}$ to be the number of tightest lowercase edges of different labels from A to C, and similarly $\sigma_{E\leftarrow C}$ the number of tightest uppercase edges of different labels from C to E. Then, we know that given a node C, $\sum_X \sigma_{X\to C} \leq n$ and $\sum_X \sigma_{X\leftarrow C} \leq n$. This is because each lowercase label has a unique start node, and each uppercase label has a unique target node, i.e. there cannot be two edges of the same label coming from different nodes to C, as shown in Figure 15.

To count the total number of triangulations, we analyze rule by rule while only using Cross-Case Composition as an example here. For Cross-Case Composition, if we look at an arbitrary triangle A, B, C under triangulation, from A to B we have $\sigma_{A\to B}$ tightest lowercase edges, and from B to C we have $\sigma_{C\leftarrow B}$ tightest uppercase edges, if we sum over all the triplets of nodes for triangulations $\sum_A \sum_B \sum_C \sigma_{A\to B} * \sigma_{C\leftarrow B} = \sum_B \sum_A \sigma_{A\to B} * (\sum_C \sigma_{C\leftarrow B}) \leq \sum_B \sum_A \sigma_{A\to B} * n \leq \sum_B n * n = n * n * n = O(n^3)$.

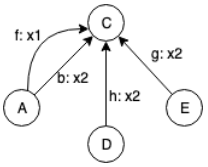

Figure 15: Illustration for $\sum_X \sigma_{X\to C} \leq n$

□

Conflict Extraction

When the network is uncontrollable, extracting conflict can explain the cause of failure and guide the adjustment of constraints. A conflict is a minimal set of constraints in the STNU that makes it uncontrollable. We describe a simple augmentation to the algorithm that supports conflict extraction, while introducing minimal overhead to the algorithm.

The procedure for conflict extraction is illustrated in Figure 16. A network is determined to be uncontrollable when the algorithm has found a semi-reducible negative cycle (line 6 and 9 in Algorithm 1), or more specifically, a negative cycle with no lower-label edges, highlighted in red in top left of Figure 16. The negative cycle may involve triangulated edges derived through triangulation, rather than from the original constraints in the network. To extract the conflict, we need to backtrack those edges to find the original constraints in the network that produced them, as shown in the bottom of Figure 16.

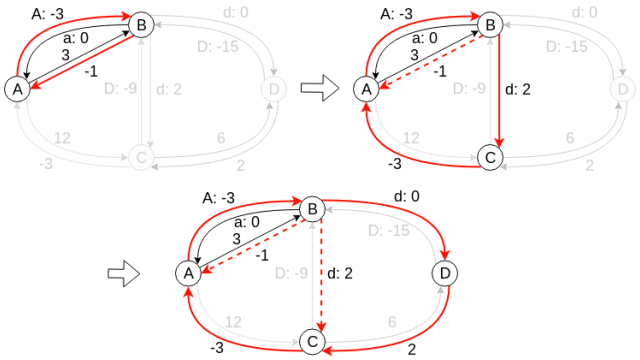

Figure 16: Rewind the semi-reducible negative cycle found during elimination into edges that come from the original constraints of the network. In this case, e_{BA} labeled with -1 has parent edges e_{BC} and e_{CA}, and e_{BC} in turn has parent edges e_{BD} and e_{DC}. The final extracted conflict is the cycle highlighted with solid red edges.

This requires a simple change to our algorithm. For each triangulation operation, we record the parent edges for the triangulated edge, i.e. the two edges that produced it accord-

ing to the triangulation rules. This means in function TRIANGULATE(e_i, e_j) in line 9 of Algorithm 3, we additionally mark the triangulated edge with: $e_{tri}.parents \leftarrow \{e_i, e_j\}$. By default, the edges in the initial LDG have an empty parent field. When a negative semi-reducible cycle is detected, EXTRACTCONFLICT, shown in Algorithm 4, will recursively unwind any triangulated edge into its parent edges, until all the edges come from original temporal constraints in the network.

Algorithm 4: EXTRACTCONFLICT

```
Input  : list of edges E
Output: Unwinded conflict cf
1 cf ← {}
2 foreach e ∈ E do
3     if e.parents then
4         cf ← cf ∪ EXTRACTCONFLICT(e.parents)
5     else
6         cf ← cf ∪ {e}
7 return cf
```

Evaluation

To compare the bucket elimination algorithm's performance with the state-of-the-art dynamic controllability checker [Morris2014], we randomly generated samples of STNUs of the following form: The STNUs are parameterized by the number of activities N_{act}, represented by contingent constraints. Additionally, pairs of activities may be constrained through their start and end events with probability of $1/(4N_{act})$, and the end event pointing to the activity with a higher index. All constraints have a lower bound of 0 and an upper bound in the range of 1 to 5 with uniform probability. For the state of the art, we tweaked the implementation from [Bhargava, Vaquero, and Williams2017], which uses Morris's DC checking algorithm [Morris2014] as its underlying algorithm. Both our algorithm and the state-of-the-art algorithm are written in Lisp and support conflict extraction.

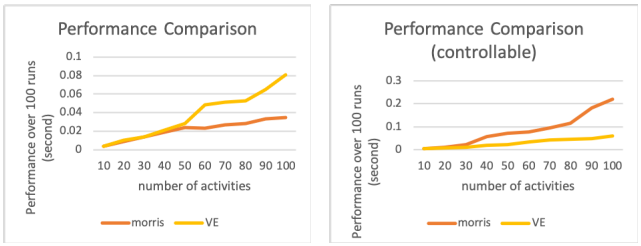

Figure 17: Performance comparison of Morris's algorithm and our bucket elimination algorithm over 100 runs. The right figure is only run on controllable STNUs.

Performance Comparison The runtime of both algorithms under varying numbers of activities from 10 to 100 is shown in Figure 17. Each case is run over 100 randomly generated STNUs. Both algorithms are reasonably fast and scale well. In the experiment shown by the figure on the left, we did not control the STNUs to be all dynamically controllable. As a result, as the number of activities increase, most of the generated STNUs are not controllable (Table 1). Interestingly, Morris's algorithm tends to be slightly faster for uncontrollable cases. When we only consider controllable cases, as shown by the figure on the right, the performance of the two algorithms only differs roughly of a constant scaling factor, with the implementation of our algorithm being slighter faster. The difference in uncontrollable cases may be caused by the fact that our bucket elimination algorithm is constrained by the canonical partial elimination order, whereas in Morris's algorithm, the negative edges are placed onto a priority queue, and the more negative the edges are, the earlier they will be examined.

N_{act}	N_{DC}	N_{act}	N_{DC}
10	58	60	2
20	32	70	1
30	24	80	0
40	9	90	0
50	7	100	0

Table 1: Number of controllable cases N_{DC} out of 100 randomly generated STNU for varying number of activities N_{act}

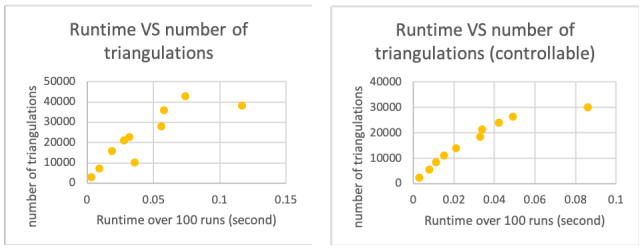

Figure 18: Run time scales with number of triangulations

Characterization with number of triangulations We are also interested in knowing whether the runtime of the algorithm roughly corresponds to the number of triangulations done by the algorithm, including both CHECKNEGCYCLE and TRIANGULATE operations.

The result is shown in Figure 18, where the experiment on the right is only tested on STNUs that are dynamically controllable. The result shows that the runtime of the algorithm is roughly proportional to the number of triangulations.

Conclusion

We drew insights from both bucket elimination algorithms and temporal network reasoning literature, and presented a bucket elimination algorithm for checking dynamic controllability that also supports conflict extraction. The algorithm is simpler to understand and implement, and achieves the same $O(n^3)$ complexity as the state-of-the-art dynamic controllability checker [Morris2014], as evidenced by our empirical results.

References

Bhargava, N., and Williams, B. C. 2019. Faster dynamic controllability checking in temporal networks with integer bounds. International Joint Conference in Artificial Intelligence.

Bhargava, N.; Vaquero, T.; and Williams, B. 2017. Faster conflict generation for dynamic controllability. In *IJCAI*, 4280–4286.

Bodlaender, H. L., and Koster, A. M. 2010. Treewidth computations i. upper bounds. *Information and Computation* 208(3):259–275.

Boerkoel, J., and Durfee, E. H. 2013. Distributed reasoning for multiagent simple temporal problems.

Cairo, M.; Hunsberger, L.; and Rizzi, R. 2018. Faster dynamic controllability checking for simple temporal networks with uncertainty. In *25th International Symposium on Temporal Representation and Reasoning (TIME 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial intelligence* 49(1-3):61–95.

Dechter, R. 1999. Bucket elimination: A unifying framework for reasoning. *Artificial Intelligence* 113(1-2):41–85.

Morris, P. 2006. A structural characterization of temporal dynamic controllability. *CP* 4204:375–389.

Morris, P. 2014. Dynamic controllability and dispatchability relationships. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, 464–479. Springer.

Nilsson, M.; Kvarnström, J.; and Doherty, P. 2016. Efficient processing of simple temporal networks with uncertainty: algorithms for dynamic controllability verification. *Acta Informatica* 53(6-8):723–752.

Shah, J. A.; Stedl, J.; Williams, B. C.; and Robertson, P. 2007. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In *ICAPS*, 296–303.

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

Supplementary Materials

Theorem 1. The triangulation rules are sound and complete in the bucket elimination algorithm for checking dynamic controllability.

Proof. We need to prove that each rule is sound and complete. For soundness, we require that the triangulation at each step will not introduce a false positive semi-reducible negative cycle. Therefore, we prove soundness by proving that if there is a semi-reducible negative cycle involving the triangulated edge, then there is indeed a corresponding semi-reducible negative cycle in the graph before elimination.

Completeness requires that eliminating parent edges will not cause false negative of semi-reducible negative cycles. Therefore, we prove completeness by proving if there is a semi-reducible negative cycle involving the parent edges in the graph before eliminating the node, then there must be a semi-reducible negative cycle with the triangulated edge in the graph after eliminating the node.

First, for rules in Figure 6, soundness directly follows from the same reduction rules from [Morris2006]. We will now show their completeness. For the Lower Label Removal and Label Removal completeness is obvious. In the following proof, we will use P_{AB} to represent a path from node A to node B. We will use $w(e)$ to denote the weight of edge e, and we will abuse the notation by writing $w(P_{AB}) = \sum_{e \in P_{AB}} w(e)$.

For Upper-Case Reduction rule, we assume by contradiction that e_{CA} and e_{DC} are involved in a semi-reducible negative cycle, but e_{DA} is not. This could only be possible if there is an edge e_{AF} with lower label b (edges labeled with lowercase b or uppercase B has to be connected to A, because A is the activated event for the received event B) in the semi-reducible negative cycle, but the same cycle is not semi-reducible when e_{CA} and e_{DC} are replaced with e_{DA}. We break it down into two cases. When $y \geq 0$, then the extension subpath that is responsible for reducing of edge e_{AF} does not involve edge e_{DC}, therefore, when replaced with e_{DA} with upper label B, the same cycle is still a semi-reducible negative cycle. When $y < 0$, then D will be eliminated before C, hence the rule will not apply in this case.

For Lower-Case Composition rule, we assume by contradiction that e_{EA} and e_{DE} are involved in a semi-reducible negative cycle, but e_{DA} is not. This could only be possible if e_{DE} is reducible but e_{DA} is not. We know that there exists P_{EF} with $w(P_{EF}) < -y$. Therefore, $w(P_{AF}) = w(P_{EF}) - w(e_{EA}) < -y - x$. Therefore, P_{AF} can also reduce e_{DA}, and we still find the same semi-reducible negatice cycle with the replaced edge.

For Cross-Case Composition rule, we need to prove for two cases. For the first case, the reasoning is similar to Upper-Case Reduction rule. We assume by contradiction that e_{EA} and e_{DE} are involved in a semi-reducible negative cycle, but e_{DA} is not. This could only be possible if there is an edge e_{AF} with lower label b in the semi-reducible negative cycle, but the same cycle is not semi-reducible when e_{EA} and e_{DE} are replaced with e_{DA}. Since $y \geq 0$, the extension subpath that is responsible for reducing of edge e_{AF} does not involve edge e_{DE}, therefore, when replaced with e_{DA} with upper label B, the same cycle is still a semi-reducible negative cycle. For the second case, the reasoning is similar to Lower-Case Composition rule. we assume by contradiction that e_{EA} and e_{DE} are involved in a semi-reducible negative cycle, but e_{DA} is not. This could only be possible if e_{DE} is reducible but e_{DA} is not. We know that there exists P_{EF} with $w(P_{EF}) < -y$. Therefore, $w(P_{AF}) = w(P_{EF}) - w(e_{EA}) < -y - x$. Therefore, since $B \neq C$, P_{AF} can also reduce e_{DA}, and we still find the same semi-reducible negatice cycle with the replaced edge.

For No-Case Reduction rule, if $x \geq 0$ and $y \geq 0$, then completeness is obvious. If $y < 0$, the rule does not apply because D will be eliminated before C. When $x < 0$ and $y \geq 0$, we assume by contradiction that e_{CA} and e_{DC} are involved in a semi-reducible negative cycle, but e_{DA} is not. It would only be possible if e_{CA} is responsible for reducing away lower case labels in the cycle that e_{DA} cannot. However, we know that for e_{CA} to reduce away any lower label edges in the cycle, the extension subpath must also include e_{DC}. This means that e_{AD} can also reduce away any lower label edges in the same cycle.

Next, we need to prove for the additional set of rules: the Lower Label Over-Approximation rules in Figure 7 that they are sound and complete.

First we prove soundness. Assume e_{DA} is in a semi-reducible negative cycle, then we have path P_{AD} with weight $w(P_{AD}) < -(x + y)$. Since at the time of elimination of E, E does not have incoming negative edges, we have $y \geq 0$. Therefore, $w(P_{AD}) < -(x + y) \leq -x$.

We also know that the start node of a contingent link e_{EB} will always be the start node of all the lowercase edges labeled with lowercase b and the target node of all the uppercase edges labeled with uppercase B as in Figure 1. Since our canonical elimination order guarantees that the start node of an negative edge will be eliminated before its end node, we know that by the time E is eliminated, all incoming negative edges, including uppercase edges labeled with B, will have been eliminated. Therefore, E has no incoming negative edges, hence no uppercase edges labeled with B is still left in the network.

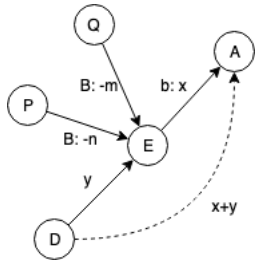

Figure 1: Elimination of E

For the first case in Figure 7, if we then replace e_{DA} with the original parent edges e_{DE} and e_{EA}, and we know that there is an extension path of P_{AD} with $w(P_{AD}) < -x$ that has no uppercase edges with label B, we know that the low-

ercase edge e_{EA} must be reducible by a prefix subpath[1] of P_{AD} or P_{AD}, according to the Lower Label Removal rule in Figure 6. Therefore, when replaced with parent edges, the cycle is still a semi-reducible negative cycle.

For the second case in Figure 7, in the same way, we have $w(P_{AD}) < -(x+y)$. if we consider P_{ED} as the edge e_{EA} extended with P_{AD}, then from the first case, we know that e_{EA} can be reduced, and we have $w(P_{ED}) < -y$. Assume by contradiction that e_{DE} cannot be reduced with P_{ED}, then there must be an edge in P_{ED} with the uppercase of the same label as e_{DE}, which is uppercase label C. Since e_{EA} does not have the same uppercase label, the uppercase labeled edge, some e_{FG}, must be in P_{AD}, and $w(P_{EF}) \geq -y$ (if not, e_{AB} can be reduced). If that is the case, we know that $w(P_{AF}) \geq -x-y$, which means that after elimination, e_{DA} cannot have been reduced by P_{AF}, and the semi-reducible negative cycle cannot exist, producing the contradiction.

For completeness, we prove that if there is a semi-reducible negative cycle involving the e_{DE} and e_{EA}, then e_{DA} is also in a semi-reducible cycle. The first case is easy to prove, if there is a semi-reducible negative cycle involving w_{DE} and w_{EA}, then $w(P_{AD}) < -(x+y)$. Therefore, after elimination of E, $w_{DA} + w(P_{AD}) < x + y - (x+y) = 0$. We still find the same semi-reducible negative cycle with the replaced edge. For the second case, since e_{DE} and e_{EA} are involved in the semi-reducible negative cycle, it means that there is a prefix subpath P_{EF} of the extension path P_{ED} with $w(P_{EF}) < -y$ and without any edge with upper case label of e_{DE}, i.e. uppercase label C. This means $w(P_{AF}) = w(P_{EF}) - w(e_{EA}) < -x - y$, and since P_{EF} does not have upper case label C, P_{AF} will not have upper case label C. Therefore, with the added edge e_{DA}, it can be reduced by P_{AF}, and we still find the same semi-reducible negative cycle with the replaced edge.

□

Theorem 2. The bucket elimination algorithm for dynamic controllability checking is sound and complete.

Proof. For soundness, at any step of the elimination, if a semi-reducible negative cycle is found and the network is deemed not dynamically controllable, then we know that the original network is also not dynamically controllable, since the reduction rules are sound and complete.

For completeness, we prove that if the network is not dynamically controllable, the algorithm will report a semi-reducible negative cycle at some point of the elimination stage. When checking controllability at any point of the elimination stage, we are only checking cycles of length 2 involving the eliminating node. If the original network is not dynamically controllable, then at any point of the elimination stage, there must be a semi-reducible negative cycle in the graph. If there is a semi-reducible negative cycle in the graph, then the termination condition for network being controllable will not hold. If the only semi-reducible negative cycle is a cycle with only negative-weight edges, it will be reported by NEXTNODE according to the termination condition. If not, a semi-reducible negative cycle will be reduced to a semi-reducible negative cycle of length 2 at some point as nodes are eliminated one by one, at which point the checking rules can correctly report the semi-reducible negative cycle. When there is finite number of nodes in the network, the algorithm will terminate. □

[1] A prefix subpath of a path is a subpath that starts from the first node of the path [Morris2006]

Theorem 3. The complexity of the bucket elimination algorithm for checking dynamic controllability is $O(n^3)$, where n is the number of nodes.

Proof. The run time of the algorithm is dominated by the total number of triangulations done throughout the process. Notice that each triplet of nodes can only go through the triangulation process once when one of the nodes is eliminated, where the edges of the triplet will be matched with the triangulation rules and the corresponding triangulation operation will occur.

We show that the number of triangulations is $O(n^3)$. Since only the tightest edges participate in the triangulation, in the following proof we only consider tightest edges between two nodes. Notice that the total number of labels (disregarding lower case or upper case) is less than n, given that there can be at most $n-1$ received events. Notice also that for all lowercase edges with the same label b, they have the same start node, and for all uppercase edges with the same label D, they have the same target node.

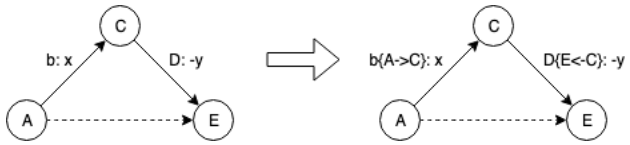

Figure 2: Relabelling the labeled edges.

To show that the number of triangulations is $O(n^3)$, we relabel the edges in the following way (Figure 2): for a lowercase edge from A to C with label b, we relabel the edge with $b_{A \to C}$, and for an uppercase edge from C to E with label D, we relabel the edge with $D_{E \leftarrow C}$. We further denote $\sigma_{A \to C}$ to be the number of tightest lowercase edges of different labels from A to C, and similarly $\sigma_{E \leftarrow C}$ the number of tightest uppercase edges of different labels from C to E. Then, we know that given a node C, $\sum_X \sigma_{X \to C} \leq n$ and $\sum_X \sigma_{X \leftarrow C} \leq n$. This is because each lowercase label has a unique start node, and each uppercase label has a unique target node, i.e. there cannot be two edges of the same label coming from different nodes to C, as shown in Figure 3.

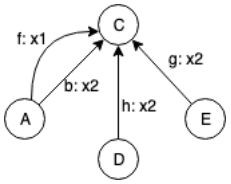

Figure 3: Illustration for $\sum_X \sigma_{X \to C} \leq n$

Now, in order to count the total number of triangulations, we analyze rule by rule:

- For Lower Label Removal and Label Removal, they can be considered as adding a constant operation after each Upper-Case Reduction, Lower-Case Composition and Cross-Case Composition operation.
- For No-Case Reduction, if we look at an arbitrary triangle A, B, C under triangulation, since there is only one tightest edge from A to B and from B to C, if we sum over all triplets of nodes for triangulations $\sum_A \sum_B \sum_C 1 * 1 = O(n^3)$
- For Upper-Case Reduction, if we look at an arbitrary triangle A, B, C under triangulation, from B to C we have $\sigma_{C \leftarrow B}$ tightest uppercase edges, and from A to B we have only one tightest edge, if we sum over all the triplets of nodes for triangulations $\sum_A \sum_B \sum_C \sigma_{C \leftarrow B} * 1 = (\sum_A 1) * (\sum_B \sum_C \sigma_{C \leftarrow B}) \leq (\sum_A 1) * (\sum_B n) = n * n * n = O(n^3)$.
- For Lower-Case Composition, if we look at an arbitrary triangle A, B, C under triangulation, from A to B we have $\sigma_{A \rightarrow B}$ tightest lowercase edges, and from B to C we have only one tightest edge, if we sum over all the triplets of nodes for triangulations $\sum_A \sum_B \sum_C \sigma_{A \rightarrow B} * 1 = (\sum_C 1) * (\sum_B \sum_A \sigma_{A \rightarrow B}) \leq (\sum_C 1) * (\sum_B n) = n * n * n = O(n^3)$.
- For Cross-Case Composition, if we look at an arbitrary triangle A, B, C under triangulation, from A to B we have $\sigma_{A \rightarrow B}$ tightest lowercase edges, and from B to C we have $\sigma_{C \leftarrow B}$ tightest uppercase edges, if we sum over all the triplets of nodes for triangulations $\sum_A \sum_B \sum_C \sigma_{A \rightarrow B} * \sigma_{C \leftarrow B} = \sum_B \sum_A \sigma_{A \rightarrow B} * (\sum_C \sigma_{C \leftarrow B}) \leq \sum_B \sum_A \sigma_{A \rightarrow B} * n \leq \sum_B n * n = n * n * n = O(n^3)$.
- For Lower Label Over-Approximation, we can consider both rules together. If we look at an arbitrary triangle A, B, C under triangulation, from A to B we have $\sigma_{A \rightarrow B}$ tightest lowercase edges, from B to C we have $\sigma_{B \rightarrow C}$ tightest lowercase edges and only one unlabeled tightest edge, if we sum over all the triplets of nodes for triangulations $\sum_A \sum_B \sum_C \sigma_{A \rightarrow B} * (\sigma_{B \rightarrow C} + 1) = (\sum_A \sum_B \sum_C \sigma_{A \rightarrow B} * \sigma_{B \rightarrow C}) + O(n^3)$. $\sum_A \sum_B \sum_C \sigma_{A \rightarrow B} * \sigma_{B \rightarrow C} = \sum_C \sum_B \sigma_{B \rightarrow C} * \sum_A \sigma_{A \rightarrow B} \leq \sum_C \sum_B \sigma_{B \rightarrow C} * n \leq \sum_C n * n = n * n * n = O(n^3)$.
  Hence, we have shown that the run time of the algorithm is $O(n^3)$.

□

References

Morris, P. 2006. A structural characterization of temporal dynamic controllability. *CP* 4204:375–389.