Faster Dynamic-Consistency Checking for Conditional Simple Temporal Networks

Luke Hunsberger*
Computer Science Department
Vassar College – Poughkeepsie, NY USA
hunsberger@vassar.edu

Roberto Posenato
Dipartimento di Informatica
Università degli Studi di Verona, Italy
roberto.posenato@univr.it

Abstract

A Conditional Simple Temporal Network (CSTN) is a structure for representing and reasoning about time in domains where temporal constraints may be conditioned on outcomes of observations made in real time. A CSTN is dynamically consistent (DC) if there is a strategy for executing its time-points such that all relevant constraints will necessarily be satisfied no matter which outcomes happen to be observed. The literature on CSTNs contains only one sound-and-complete DC-checking algorithm that has been implemented and empirically evaluated. It is a graph-based algorithm that propagates labeled constraints/edges. A second algorithm has been proposed, but not evaluated. It aims to speed up DC checking by more efficiently dealing with so-called negative q-loops.

This paper presents a new two-phase approach to DC-checking for CSTNs. The first phase focuses on identifying negative q-loops and labeling key time-points within them. The second phase focuses on computing (labeled) distances from each time-point to a single sink node. The new algorithm, which is also sound and complete for DC-checking, is then empirically evaluated against both pre-existing algorithms and shown to be much faster across not only previously published benchmark problems, but also a new set of benchmark problems. The results show that, on DC instances, the new algorithm tends to be an order of magnitude faster than both existing algorithms.

Introduction

A Conditional Simple Temporal Network (CSTN) is a data structure for reasoning about time in domains where some constraints may apply only in certain scenarios. For example, a patient who tests positive for a certain disease may need to receive care more urgently than someone who tests negative. Conditions in a CSTN are represented by propositional letters whose truth values are not controlled, but instead observed in real time. Just as doing a blood test generates a positive or negative result that is only learned in real time, the execution of an observation time-point in a CSTN generates a truth value for its corresponding propositional letter. An execution strategy for a CSTN specifies when its time-points will be executed. A strategy can be dynamic in that its decisions can react to information from past observations. A CSTN is said to be dynamically consistent (DC) if it admits a dynamic strategy that guarantees the satisfaction of all relevant constraints no matter which outcomes are observed during execution. Cairo and Rizzi (2016) showed that the DC-checking problem for CSTNs is PSPACE-complete.

Different varieties of the DC property have been defined that differ in how reactive a strategy can be. Tsamardinos et al. (2003) stipulated that a strategy can react to an observation after any arbitrarily small, positive delay. Comin et al. (2015) defined ϵ-DC, which assumes that a strategy’s reaction times are bounded below by a fixed ϵ > 0. Finally, Cairo et al. (2016; 2017a) defined π-DC, which allows a dynamic strategy to react instantaneously (i.e., after zero delay).

This paper focuses exclusively on the π-DC property. Cairo et al. (2016) presented the first sound-and-complete π-DC-checking algorithm. However, that algorithm, which is (pseudo) singly-exponential in the number of observation time-points, has never been implemented or empirically evaluated. Hunsberger and Posenato (2018) presented an alternative algorithm (which we shall call HP18), based on the propagation of labeled constraints. They empirically evaluated the algorithm, demonstrating its practicality. Later, noting that the HP18 algorithm can get bogged down, repeatedly cycling through graphical structures called negative q-loops, Hunsberger and Posenato (2019) presented (what we shall call) the HP19 algorithm, which included a rule that can generate edges labeled by expressions such as (−∞, α], and generalized existing rules to accommodate such edges. They did not empirically evaluate the HP19 algorithm, but conjectured that it would deal more effectively with negative q-loops.

This paper presents a new approach to π-DC-checking for CSTNs that involves two phases. The first phase focuses on identifying negative q-loops and properly labeling key time-points—not edges—within such loops. The second phase focuses on computing (labeled) distances from each time-point to a single sink node. The new algorithm, which is also sound and complete for DC-checking, is then empirically evaluated against both pre-existing algorithms and shown to be much faster across not only previously published benchmark problems, but also a new set of benchmark problems. The results...
show that, on DC instances, the new algorithm tends to be an order of magnitude faster than both existing algorithms. On all other benchmark cases, the new algorithm performs better than or equivalently to the existing algorithms.

Preliminaries

Dechter et al. (1991) introduced Simple Temporal Networks (STNs) to facilitate reasoning about time. An STN has real-valued variables, called time-points, and binary difference constraints on those variables. Most STNs have a time-point whose value is fixed at zero. A consistent STN is one that has a solution as a constraint satisfaction problem.

Tsamardinos et al. (2003) presented CSTNs, which augment STNs to include observation time-points and their associated propositional letters. In a CSTN, the execution of an observation time-point generates a truth value for its associated letter. In addition, each time-point can be labeled by a conjunction of literals specifying the scenarios in which that time-point must be executed. Finally, they noted that for any reasonable CSTN, the propositional labels on its time-points must satisfy certain properties.

Hunsberger et al. (2012) generalized CSTNs to include labels on constraints, and formalized the properties held by well-defined CSTNs. Then Cairo et al. (2016) showed that for well-defined CSTNs, no loss of generality results from removing the labels from its time-points. Therefore, this paper restricts attention to CSTNs whose time-points have no labels—the so-called streamlined CSTNs—and henceforth uses the term CSTN to refer to streamlined CSTNs.

Fig. 1 shows two sample CSTNs in their graphical forms, where nodes represent time-points, and labeled, directed edges represent conditional binary difference constraints. For example, in the top figure, Z = 0; and P?, Q?, and W? are observation time-points whose execution generates truth values for p, q, and w, respectively. The edge from Y to Q? being labeled by (2, pw) indicates that the constraint Q? − Y ≤ 2 applies only in scenarios where p and w are both true.

The Dynamic Consistency of CSTNs

Since the execution of an observation time-point generates a truth value for its associated letter, a dynamic execution strategy can react to observations, in real time, possibly making different execution decisions in different scenarios. A dynamically consistent CSTN is one that has an execution strategy that guarantees that all relevant constraints will be satisfied no matter which values are observed in real time. This paper focuses on π-dynamic strategies, which can react instantaneously to observations (Cairo, Comin, and Rizzi 2016). The full set of definitions is given in the Appendix.

Figure 1: Two sample CSTN graphs

Existing π-DC-Checking Algorithms

This paper restricts attention to the π-DC-checking problem for CSTNs (i.e., execution strategies can react instantaneously to observations). For convenience, we use the term DC to mean π-DC. The π-DC-checking algorithms discussed in this paper are all based on the propagation of labeled constraints. In graphical terms, each algorithm employs a set of rules for generating new edges from existing edges in the CSTN graph. Whereas the characteristic feature of an inconsistent STN is the existence of a negative-length loop, the characteristic feature of a non-DC CSTN is the existence of a negative-length loop whose edges have mutually consistent propositional labels. For example, a CSTN with the loop shown below

must be non-DC since in any scenario consistent with pqr, both constraints along the negative loop must be satisfied, which is impossible. (In other networks, such a loop might only be revealed after extensive constraint propagation.) However, a DC CSTN may contain negative-length loops whose edges have mutually inconsistent propositional labels; they are called negative q-loops. For example, the CSTN at the top of Fig. 1 is DC, despite having two negative q-loops: one from X to P? to X, and one from Y to Q? to Y. (Note, for example, that the label pw on the edge from Y to Q? is inconsistent with the label ¬p¬w on the edge from Q? to Y.)

In this network, propagations involving the negative q-loops cannot lead to a negative loop with a consistent label; hence, the network is non-DC. However, negative q-loops are not always benign. For example, propagating the negative q-loops in the CSTN at the bottom of Fig. 1 will eventually generate a negative loop with a consistent label, implying that that network is not DC. For these reasons, negative q-loops pose difficult challenges for any π-DC-checking algorithm.

Each algorithm in this paper generates new edges in the CSTN graph until: (1) a negative-length self-loop (i.e., a negative-length edge from a node to itself) with a consistent label is generated, or (2) no new edges can be generated. In case (1), the network is not DC; in case (2), it is DC.

The HP18 Algorithm

The only sound-and-complete π-DC-checking algorithm that has been implemented and empirically evaluated in the literature is the π-DC-Check algorithm of Hunsberger and Posenato (2018), hereinafter called HP18. To deal with constraints having inconsistent labels, the algorithm sometimes generates a new kind of propositional label, called a q-label.

Definition 1 (Q-literals, q-labels). A q-literal has the form ?p, where p is a propositional letter. A q-literal represents that a proposition’s value is unknown. A q-label is a conjunction of literals and/or q-literals. Q∗ denotes the set of all q-labels.

For example, p(?q)¬r and (?p)(?q)(?r) are both q-labels. The operator extends conjunction to accommodate q-labels. Intuitively, if the constraint C1 is labeled by p, and C2 is labeled by ¬p, then both C1 and C2 must hold as long as the value of p is unknown, represented by p ∗ ¬p = ?p.
Table 1: Edge-generation rules used by the HP$_{18}$ algorithm

<table>
<thead>
<tr>
<th>Rule</th>
<th>Edge Generation</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{R}_{1}^{a}$</td>
<td>$X \xrightarrow{(u,\alpha)} W \xrightarrow{(v,\beta)} Z$</td>
<td>$u + v &lt; 0$ and $\alpha \beta \in \mathcal{P}^*$</td>
</tr>
<tr>
<td>$\mathcal{R}_{1}^{b}$</td>
<td>$P? \xrightarrow{(v,\alpha)} Z$</td>
<td>$w &lt; 0$, $\pi p \notin \alpha \in \mathcal{Q}^*$</td>
</tr>
<tr>
<td>$\mathcal{R}_{1}^{c}$</td>
<td>$P? \xrightarrow{(\pi,\alpha)} Z$</td>
<td>$w &lt; 0$, $\pi p \notin \alpha \in \mathcal{Q}^*$</td>
</tr>
</tbody>
</table>

$W, X, Y \in \mathcal{T}$; $Z = 0$; $P? \in \mathcal{O} \mathcal{T}$; and $u, v, w \in \mathbb{R}$.

Table 2: Edge-generation rules used by the HP$_{19}$ algorithm shown in Fig. 2) to generate negative loops with consistent labels, at which point the algorithm will correctly conclude that the network is not DC. However, although the CSTM at the top of Fig. 1 has a similar structure, the presence of $W$? and constraints labeled by $u$ and $w$ combine to ensure that it is DC, which the HP$_{19}$ algorithm will discover after cycling through the negative q-loops hundreds of times.

The HP$_{19}$ Algorithm

Aiming to speed up $\pi$-DC checking by dealing more effectively with negative q-loops, Hunsberger and Posenato (2019) introduced a new set of sound-and-complete edge-generation rules which, in this paper, we call the HP$_{19}$ algorithm. It begins with the $\mathcal{R}_{0}^{a}$ rule shown in Table 2, that covers a special case of labeled propagation in which the two edges (from $X$ to $Y$) form a negative q-loop. (If $\alpha \beta = \alpha \beta \in \mathcal{P}^*$, then the CSTM can be immediately rejected as not DC.) They showed that instead of setting the weight on the generated loop to $u + v < 0$, it is sound to set it to $-\infty$. Intuitively, such a loop can be understood as saying that $X$ cannot be executed as long as the label $\alpha \beta$ is (or might yet be) true. For example, a loop from $X$ to $X$ labeled by $(\neg, \neg q)q$ represents that $X$ cannot be executed as long as $p$ is unknown and $q$ is (or might yet be) true. They showed that the $\mathcal{R}_{0}^{a}$ rule can greatly speed up $\pi$-DC checking because instead of repeatedly cycling through negative q-loops many times, the HP$_{19}$ algorithm may cycle through them only once, using the rest of the rules from Table 2, which are straightforward extensions of the corresponding rules from Table 1 to accommodate $-\infty$, and to generate edges pointing at any node—not just $Z$. In addition, the $\mathcal{R}_{2}^{b}$ rule can generate q-labeled edges, and the $\mathcal{R}_{0}^{b}$ rule can be incorporated into the $\mathcal{R}_{2}^{b}$ rule as a post-process. Note, too, that a $-\infty$ value generated by the $\mathcal{R}_{0}^{b}$ rule can be propagated by $\mathcal{R}_{2}^{b}$, since $X$ or $Y$ may be identical to $W$.

In Hunsberger and Posenato (2018), the $\mathcal{R}_{0}^{a}$, $\mathcal{R}_{1}^{a}$ and $\mathcal{R}_{1}^{b}$ rules were called LP$_2$, qR$_0$ and qR$_1$, respectively. We use the $\mathcal{R}_{0}^{a}$, $\mathcal{R}_{1}^{a}$, $\mathcal{R}_{1}^{b}$ notation throughout the paper to highlight the similarities among groups of rules, while keeping the notation manageable. For Tables 1-3, the subscript specifies the number of the table in which the rule first appears; the superscript specifies the general class to which the rule belongs: $a$ for generalized constraint propagation, $b$ for basic label modification, and $c$ for complex label modification.

Definition 2 (\$\star\$). The operator, $\star$: $\mathcal{Q}^* \times \mathcal{Q}^* \rightarrow \mathcal{Q}^*$, is defined thusly. First, for any $p \in \mathcal{P}$, $p \star p = p$ and $\neg p \star \neg p = \neg p$. For all other combinations of $p_1, p_2 \in \{p, \neg p, \neg \neg p\}$, $p_1 \star p_2 = \neg p$. Finally, for any q-labels $\ell_1, \ell_2 \in \mathcal{Q}^*$, $\ell_1 \star \ell_2 \in \mathcal{Q}^*$ denotes the conjunction obtained by applying $\star$ in pairwise fashion to matching literals from $\ell_1$ and $\ell_2$, and conjoining any unmatched literals.

For example: $(\neg q)(\neg r) \star (q r) = (p q)(\neg r)\neg st$.

The HP$_{19}$ algorithm is sound and complete for $\pi$-DC checking, it can get bogged down cycling through negative q-loops. For example, recall the CSTN from the bottom of Fig. 1, a portion of which is shown in Fig. 2. It shows ten applications of the $\mathcal{R}_{0}^{a}$, $\mathcal{R}_{1}^{a}$ and $\mathcal{R}_{1}^{b}$ rules, generating the dashed edges in the order indicated by the parentheses numbers, the end result of which is that the weights on the edges from $P$ to $Z$, and $Q$ to $Z$ have changed from $-13$ to $-15$. After cycling through these intersecting negative q-loops several hundred more times, the resulting edges will combine with the upper-bound edges from $Z$ to $P$? and $Z$ to $Q$? (not
where weights on edges could be piecewise-linear functions.)

(Their only intent was to show its usefulness in a context

Fig. 3 shows that the rules from Table 2 only pass through

negative q-loops from Fig. 2 once to generate uncondi-
tional lower bounds of $\infty$ for $Q$, and $P$, at Steps (7) and

(8), respectively. Since $P$ and $Q$ have upper bounds of 500
(cf. the bottom of Fig. 1), the network must be non-DC. The

network from the top of Fig. 1 can be similarly analyzed, ex-
ting lower bounds of $\infty$ for $Q$, and $P$, end up being conditioned on $?w$. The $R_2^q$ rule, using the edge

from $W$ to $Z$, then generates unconditional lower bounds of 300 for $Q$, and $P$, enabling the network to be DC.

Although expected to outperform the $HP_{18}$ algorithm

on networks with negative q-loops, Hunsberger and Pope-
nato (2019) did not empirically evaluate the $HP_{19}$ algorithm.

(Their only intent was to show its usefulness in a context

where weights on edges could be piecewise-linear functions.)

**HP$_{20}$: A Faster $\pi$-DC-Checking Algorithm**

This section introduces a new $\pi$-DC-checking algorithm for

CSTNs, called HP$_{20}$, that builds on the algorithms seen above.

The primary insight is that the semantics of satisfying an edge

labeled by $(-\infty, \alpha)$, for some $\alpha \in Q^* \backslash P^*$ does not depend

on the target node of the edge, but only on its source node.

As a result, much of the propagation of such labeled values by

the HP$_{19}$ algorithm is redundant. The HP$_{20}$ algorithm avoids

this problem by associating such labeled values only with

nodes, not edges. The HP$_{20}$ algorithm also separates the job

of finding negative q-loops, which it does in a pre-processing

phase, from the main algorithm.

**The Semantics of Constraints on Nodes**

Hunsberger and Popezato (2018) defined the semantics

of satisfying a (lower-bound) q-labeled constraint $X \xrightarrow{\delta, \alpha} Z$ for

any $\delta < 0$ and $\alpha \in Q^*$. Applying this definition to cases

where $\delta = -\infty$, and letting the target node be any $Y \in T$,

yields the following (Hunsberger and Popezato 2019).

**Definition 3.** The execution strategy $\sigma$ satisfies the q-labeled

constraint $X \xrightarrow{(-\infty, \alpha)} Y$ if for each scenario $s$:

1. $|\sigma(s)|_X \geq |\sigma(s)|_Y + \infty$; or

2. some $\bar{a} \in \{a, -a, ?a\}$ appears in $\alpha$ such that $\sigma(s)$ ob-

serves a $\pi$-before $Y$ and $s \neq \bar{a}$.4

Since clause (1) cannot be satisfied, it follows that $\sigma$ can

only satisfy such a constraint if $\sigma(s)$ does not execute $X$

until it first executes some observation time-point $A'$ that

generates a value for $a$ that ensures that $s \neq \bar{a}$. The critical

point is that such a constraint only applies to the source node

$X$; it does not involve $Y$ at all. For this reason, it makes

sense to associate such a constraint to the node $X$ (e.g., as

in $X_{(-\infty, \alpha)}$, not to the edge from $X$ to $Y$). Furthermore, it

would be pointless to forward-propagate such constraints,

because the resulting edge would have the same source node,

and hence would be redundant.

**Finding Negative Q-Loops**

The NQLFinder algorithm, shown in Algorithm 1, is a pre-

process that uses the rules listed in Table 3 to find all negative

q-loops having at most $n = |T|$ time-points.5 The $R^q_3$ rule

propagates forward from each source node $X$, generating

negative-length edges, but note that $\nu$ (i.e., the length of the

second edge in the rule) may be non-negative. The $R^{q\alpha}_3$ rule is

Algorithm 1: NQLFinder ($G$)

```
Input: CSTN $G = (T, E)$.
Output: $G$ modified by NQLF rule.
Q := E, newQ := $\emptyset$, n := $|T| - 1$
while Q $\neq \emptyset$ and n $>$ 0 do
  while Q $\neq \emptyset$ do
    (X, Y) := extract an edge from Q
    foreach (Y, W) $\in$ E do
      if (X, W) $\in$ E and n $\neq |T| - 1$ then
        continue // Update (X, W) only once
        eXWfilled := NQLF ((X, Y), (Y, W))
      if eXWfilled is new or modified then
        newQ = newQ $\cup$ eXWfilled
  n := n $-$ 1
  Q = newQ
```

(2) some $\bar{a} \in \{a, -a, ?a\}$ appears in $\alpha$ such that $\sigma(s)$ ob-

serves a $\pi$-before $Y$ and $s \neq \bar{a}$.4

5A negative q-loop with more than $n$ time-points must have a sub-loop that is a negative q-loop with at most $n$ time-points.
similar to the $R_2^\oplus$ rule from Table 2, except that it generates a labeled value associated with the node $X$, not the edge from $X$ to $X$. The $R_2^\oplus$ and $R_3^\oplus$ rules are used as post-processes for $R_0^\oplus$ and $R_3^\oplus$, respectively, to remove instances of any $\tilde{v} \in \{p, \neg p, \top\}$ when $X$ is the corresponding observation time-point $P^\oplus$. In the implementation, the four rules from Table 3 are folded into a single composite rule, called NQLF.

The overall aim of the NQLFinder algorithm is to find all nodes that can be labeled by $\langle -\infty, \alpha \rangle$ for some $\alpha$. (A single node may have a set of such labels, each with a different $\alpha$.) Often, not every node in a negative q-loop can be so labeled (e.g., source nodes of non-negative-length edges). When done, any edges discovered by the NQLFinder algorithm are discarded; only the node constraints are kept.

When NQLFinder is run on the CSTN at the bottom of Fig. 1, single applications of the $R_3^\oplus$ rule generate labels of $\langle -\infty, \top \rangle$ for $Q^\oplus$, and $\langle -\infty, \bot \rangle$ for $P^\oplus$. Afterward, the main algorithm, discussed below, can use rules $R_0^\oplus$ and $R_3^\oplus$ from Table 4 to generate the unconditional lower bounds of $\infty$ on $P^\oplus$ and $Q^\oplus$ which, given their finite upper bounds, implies that the network must be non-DC.

### Propagating Constraints

The main part of the $\text{HP}_20$ algorithm uses the rules shown in Table 4.6 Like the $\text{HP}_19$ rules from Table 1, all edges generated by the $\text{HP}_20$ rules have $Z$ as their target, and have finite numerical weights. Like the $\text{HP}_19$ rules from Table 2, the $\text{HP}_20$ rules generate labels such as $\langle -\infty, \alpha \rangle$; however, such labels are applied to nodes, not edges. The $R_3^\oplus$ rule is identical to the $R_4^\toplus$ rule used by the $\text{HP}_19$ algorithm, except that the $R_4^\oplus$ rule accommodates q-labels. Each instance of the $R_3^\oplus$ rule propagates a $\langle -\infty, \beta \rangle$ label on a node backward across an edge to generate a new node label. The $R_0^\oplus$ rule is the same as the one used by NQLFinder (cf. Table 3). The $R_0^\oplus$, $R_3^\oplus$, and $R_3^\oplus$ rules extend the $R_4^\oplus$ rule to accommodate $\langle -\infty, \alpha \rangle$ labels on nodes in different positions.

Since all edges manipulated by the $\text{HP}_20$ algorithm have $Z$

---

Table 3: Edge-generation rules for NQLFinder

<table>
<thead>
<tr>
<th>Rule</th>
<th>Edge Generation</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_0^\oplus$</td>
<td>$X \xrightarrow{(u, \alpha)} W \xrightarrow{(u + v, \alpha \ast \beta)} Y$</td>
<td>$(u &lt; 0$ and $u + v &lt; 0$) or $(\alpha \ast \beta \in P^\oplus)$</td>
</tr>
<tr>
<td>$R_3^\oplus$</td>
<td>$X \xrightarrow{(u, \alpha)} W \xrightarrow{(u \ast \alpha \ast \beta)} Z$</td>
<td>$u &lt; 0$, $u + v &lt; 0$, and $\alpha \ast \beta \in Q^\oplus$</td>
</tr>
<tr>
<td>$R_3^\oplus$</td>
<td>$P^\bot \xrightarrow{(u, \alpha \ast \beta)} X$</td>
<td>$w &lt; 0$, $\hat{p} \notin \alpha \in Q^\oplus$</td>
</tr>
<tr>
<td>$R_3^\oplus$</td>
<td>$P^\bot \xrightarrow{(u, \alpha \ast \beta)} X$</td>
<td>$\hat{p} \notin \alpha \in Q^\oplus$</td>
</tr>
</tbody>
</table>

Table 4: Edge-generation rules for the $\text{HP}_20$ algorithm

<table>
<thead>
<tr>
<th>Rule</th>
<th>Edge Generation</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$R_4^\oplus$</td>
<td>$X \xrightarrow{(u, \alpha)} W \xrightarrow{(u + v, \alpha \ast \beta)} Z$</td>
<td>$(\alpha \ast \beta = \alpha \beta \in P^\oplus)$ or $(u &lt; 0$ and $u + v &lt; 0$)</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$X \xrightarrow{(u, \alpha \ast \beta)} W \xrightarrow{(\hat{u}, \alpha \ast \beta)} Y$</td>
<td>$u &lt; 0$, $\alpha \ast \beta \in Q^\oplus \setminus P^\oplus$</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$P^\bot \xrightarrow{(w, \alpha \ast \beta)} Y$</td>
<td>$w &lt; 0$, $\hat{p} \notin \alpha \ast \beta \in Q^\oplus$</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$P^\bot \xrightarrow{(u, \alpha \ast \beta)} Y$</td>
<td>$u &lt; 0$</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$P^\bot \xrightarrow{(u, \alpha \ast \beta)} Y$</td>
<td>$u &lt; 0$</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$P^\bot \xrightarrow{(w, \alpha \ast \beta)} Y$</td>
<td>$w &lt; 0$, $\hat{p} \notin \alpha \ast \beta \in Q^\oplus$</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$P^\bot \xrightarrow{(u, \alpha \ast \beta)} Y$</td>
<td>$u &lt; 0$</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$P^\bot \xrightarrow{(w, \alpha \ast \beta)} Y$</td>
<td>$w &lt; 0$, $\hat{p} \notin \alpha \ast \beta \in Q^\oplus$</td>
</tr>
<tr>
<td>$R_9^\oplus$</td>
<td>$P^\bot \xrightarrow{(u, \alpha \ast \beta)} Y$</td>
<td>$u &lt; 0$</td>
</tr>
</tbody>
</table>

In summary, unlike all previous algorithms, Algorithm 2 does not add any edges to the network and checks the dynamic consistency by determining the minimal distance to each node in relevant scenarios. This approach avoids a large amount of redundant propagation of labeled values on edges that is done by earlier algorithms.

### Experimental Evaluation

This section compares the performance of our new $\text{HP}_20$ algorithm against the pre-existing $\text{HP}_19$ and $\text{HP}_19$ algorithms. $\text{HP}_20$ refers the implementation of Algorithm 2; $\text{HP}_19$ is the freely available implementation of the $\pi$-DC-checking algorithm (Hunsberger and Posenato 2018); $\text{HP}_19$ is our implementation of the alternative $\pi$-DC-checking algorithm proposed by Hunsberger and Posenato (2019). All algorithms and procedures were implemented in Java and executed on a JVM 8 in a Linux box with two AMD Opteron 4334 CPUs and 64GB of RAM. The implementation of all algorithms
Algorithm 2: $HP_{20}$ ($G$)

Input: CSTN $G = (T, E)$
Output: Consistency status: YES/NO

Z.d := \{(0, \emptyset)\} // Z = first node; v.d is v’s potential
NQLFinder($G$) // Generate ($-\infty, \alpha$) values

Q := \{Z\}
while Q \neq \emptyset do

QObs := \{

while Q \neq \emptyset do // Update node distances

X := extract a node from Q
foreach eYX := (Y, X) \in E do
foreach (u, \alpha) \in X.d do

if Y.d potential was updated then

\text{Insert Y in Q}
if Y is an observation time point then

\text{Insert Y in QObs}

// Apply pot° among obs. time-points ONLY
QObs1 := QObs
while QObs1 \neq \emptyset do

A! := extract a node from QObs1
foreach observation time-point X? \in V do

// Apply pot° to X w.r.t A?
foreach (u, \alpha) \in X.d do

if \gamma \in P^x and u.d = -\infty then return NO
X.d(\gamma) := u

if X.d potential was updated then

\text{Insert X in Q, QObs and in QObs1}

// Apply pot° to other time-points
foreach observation time-point A? \in QObs do

X := extract a node from Q
foreach eYX := (Y, X) \in E do

if X is an obs. time-point then continue
// Apply pot° to X w.r.t A?
foreach (u, \alpha) \in X.d do

if \gamma \in P^x and u.d = -\infty then return NO
X.d(\gamma) := u

if X.d potential was updated then

\text{Insert X in Q}

return YES

\end{algorithm}

and procedures is freely available as Java Package (Posenato 2019).

All implementations were tested on instances of the four benchmarks from Hunsberger and Posenato (2016). Each benchmark has at least 250 DC and 250 non-DC CSTNs, obtained from random workflow schemata generated by the ATAPIS toolset. The numbers of activities ($N$) of random workflows and choice connectors (corresponding to CSTN observations ($|P|$)) were varied, as shown in Fig. 4.

We fixed a time-out of 10 minutes (m) for the execution of each algorithm on each instance. For the DC instances, $HP_{19}$ timed out on 32 of the 250 instances, while $HP_{18}$ timed out on only 3. Most of time-outs occurred in benchmark 4. For the non-DC instances, $HP_{19}$ timed out on 2 of the 250 instances, while $HP_{18}$ timed out on 18. The $HP_{20}$ algorithm never timed out.

Fig. 4 displays the average execution times of the three algorithms across all eight benchmarks (4 for DC instances, 4 for non-DC instances), each point representing the average execution time for instances of the given size. The size of the benchmarks allows the determination of 95% confidence intervals for the results. The results demonstrate that the $HP_{18}$ and $HP_{19}$ algorithms perform differently for different kinds of networks: $HP_{18}$ is better than $HP_{19}$ when instances are DC, while $HP_{19}$ is better when instances are not DC. The reason is that $HP_{18}$ generates labeled values only on edges pointing at Z, while $HP_{19}$ can generate labeled values for any edge. Therefore, when instances are DC (i.e., no negative cycle with a consistent label), the propagations are exhausted earlier by the $HP_{18}$ algorithm. In contrast, when an instance is non-DC, $HP_{19}$ tends to detect the negative loop with a consistent label much more quickly, due to its more efficient processing of negative q-loops. (The $HP_{18}$ algorithm can cycle repeatedly through negative q-loops until some upper bound is violated, which can take a long time if the upper bound is relatively large.)

The $HP_{20}$ algorithm can be viewed as combining the
and each containing just 1 proposition. The benchmark is and some negative q-loops with 6 edges, cycle weight -1, DC, 150 non-DC) each having 100 nodes, 7 propositions, edge weights, number of observation time-points in q-loops, circuit weight of negative q-loops, minimal and maximal negative q-loops, number of edges in negative q-loops, the specific features. Some features can be given as input to the possible to generate random instances having a variety of new random generator of CSTN instances by which it is to the structure of possible CSTN instances, we set up a HP performance is better than both of the other algorithms when HP the node potentials more efficiently than HP q-loops efficiently, and the main HP non-DC, the NQLFinder pre-process can detect negative q-loops more efficiently as a pre-process. Second it algorithm can manage strengths of the HP18 and HP19 algorithms. First, it identifies negative q-loops more efficiently as a pre-process. Second it uniformly treats all constraints as labeled potentials on nodes, avoiding the redundant propagations of \((-\infty, \alpha)\) values on edges by HP19. Since it updates only the potentials of nodes, when an instance is DC, it updates such potentials similarly to how the HP18 algorithm updates edges pointing at Z, without any other useless computations. When an instance is non-DC, the NQLFinder pre-process can detect negative q-loops efficiently, and the main HP20 algorithm can manage the node potentials more efficiently than HP19. Therefore, its performance is better than both of the other algorithms when instances are positive, while it is more or less equivalent to the HP19 algorithm when instances are negative.

To study the behavior of the three algorithms with respect to the structure of possible CSTN instances, we set up a new random generator of CSTN instances by which it is possible to generate random instances having a variety of specific features. Some features can be given as input to the random generator: number of nodes, number of propositions, probability of an edge for each pair of nodes, minimal number of negative q-loops, number of propositions appearing in negative q-loops, number of edges in negative q-loops, the circuit weight of negative q-loops, minimal and maximal edge weights, number of observation time-points in q-loops, minimal distance from observation time-points to Z, etc.

Then, we built two new benchmarks. The first, 100n7pQL6nQL1pQL, contains 300 random instances (150 DC, 150 non-DC) each having 100 nodes, 7 propositions, and some negative q-loops with 6 edges, cycle weight -1, and each containing just 1 proposition. The benchmark is divided into 3 sub-benchmarks of 50 instances each: the first contains instances in which at least 2 negative q-loops are present, the second contains instances having at least 4 negative q-loops, and the third contains instances having at least 6 negative q-loops. In each instance, the weight of an edge is a random value in \([-150, 150]\). Figure 5 depicts the execution times of the three algorithms on the 100n7pQL6nQL1pQL benchmark. The time-out was fixed to 15 m.

For DC instances, HP19 timed out on approx. 43% of the instances, while HP18 and HP20 never timed out. For non-DC instances, HP19 timed out on approx. 6% of the 150 instances, HP18 for approx. 51%, and HP20 for approx. 1%. These results confirm that instances having negative q-loops are harder to solve than those without negative q-loops. Overall, the HP20 algorithm performs best across both DC and non-DC instances. The results also suggest that the difference among having 2, 4, and 6 negative q-loops does not significantly affect the execution times for any of the algorithms.

The second benchmark, 100n7pQL6nQL1pQLFarObs, contains the same instances as the first benchmark, but with the distances of observation time-points from Z modified. Each observation time-point has an edge to Z with a random value (distance) in the range \([-450, -300]\). In this way, we wanted to study how the algorithms work for solving negative q-loops. Figure 6 depicts the execution times of the three algorithms on the 100n7pQL6nQL1pQLFarObs benchmark. The time-out was fixed to 15 m. For the DC instances, HP19 timed out for approx. 36% of the 150 instances, while HP18 and HP20 never timed out. For the non-DC instances, HP19 timed out for approx. 8% of the 150 instances, while HP18 for approx. 59%, and HP20 for approx. 4%. Although we had expected the HP18 algorithm to perform much worse on the non-DC instances in this benchmark, the results did not confirm this. We will explore different benchmarks to further understand the different behaviors of the three algorithms.

The main takeaway from our evaluation is that the HP20 algorithm performs significantly better than the existing algorithms across many types of benchmarks, and always performs at least as well as those algorithms on all benchmarks.

**Conclusions**

This paper presented a new approach to DC checking for CSTNs that results in a sound-and-complete algorithm, called HP20, that is empirically demonstrated to be significantly faster than pre-existing DC-checking algorithms across not only existing benchmarks, but also across a new set of benchmarks. The HP20 algorithm more efficiently identifies important graphical structures called negative q-loops and more efficiently manages the propagation of labeled values of the form \((-\infty, \alpha)\). In addition, unlike previous algorithms, the main phase of the new algorithm only updates labeled values—whether finite or infinite—on nodes, not edges.

Looking forward, we plan to evaluate the HP20 algorithm across a wider variety of benchmark problems to determine which graphical features most significantly impact its performance.
The truth values of propositions in a CSTN are not known in advance, but a π-dynamic execution strategy can react instantaneously to observations. To make instantaneous reactivity plausible, a π-execution strategy must specify an order of dependence among simultaneous observations.

Definition 4 (Labels). Let \( \mathcal{P} \) be a set of propositional letters. A label is a conjunction of (positive or negative) literals from \( \mathcal{P} \). The empty label is notated \( \square \); and \( \mathcal{P}^* \) denotes the set of all satisfiable labels with literals from \( \mathcal{P} \).

Definition 5 (CSTN). A Conditional Simple Temporal Network (CSTN) is a tuple, \( (\mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O}) \), where:

- \( \mathcal{T} \) is a finite set of real-valued time-points (i.e., variables);
- \( \mathcal{P} \) is a finite set of propositional letters (or propositions);
- \( \mathcal{C} \) is a set of labeled constraints, each having the form, \( (Y - X \leq \delta, \ell) \), where \( X, Y \in \mathcal{T}, \delta \in \mathbb{R}, \) and \( \ell \in \mathcal{P}^* \);
- \( \mathcal{OT} \subseteq \mathcal{T} \) is a set of observation time-points (OTPs); and
- \( \mathcal{O} : \mathcal{P} \rightarrow \mathcal{OT} \) is a bijection that associates a unique observation time-point to each propositional letter.

In a CSTN graph, the observation time-point for \( p \) (i.e., \( \mathcal{O}(p) \)) is usually denoted by \( P? \); and each labeled constraint, \( (Y - X \leq \delta, \ell) \), is represented by an arrow from \( X \) to \( Y \), annotated by the labeled value \( \delta, \ell \). If \( \ell \) is empty, then the arrow is labeled by \( \delta \), as in an STN graph. Since \( X \) and \( Y \) may participate in multiple constraints of the form, \( (Y - X \leq \delta, \ell) \), the edge from \( X \) to \( Y \) may have multiple labeled values of the form, \( (\delta, \ell) \).

Definition 6 (Schedule). A schedule for a set of time-points \( \mathcal{T} \) is a mapping, \( \psi : \mathcal{T} \rightarrow \mathbb{R} \). The set of all schedules for any subset \( \mathcal{T} \) is denoted by \( \Psi \).

Definition 7 (Scenario). A function, \( s : \mathcal{P} \rightarrow \{\top, \bot\} \), that assigns a truth value to each \( p \in \mathcal{P} \), is called a scenario. For any label \( \ell \in \mathcal{P}^* \), the truth value of \( \ell \) determined by \( s \) is denoted by \( s(\ell) \). \( \mathcal{I} \) denotes the set of all scenarios over \( \mathcal{P} \).

The projection of a CSTN onto a scenario, \( s \), is the STN obtained by including only the constraints whose labels are true under \( s \) (i.e., that must be satisfied in that scenario).

Definition 8 (Projection). Let \( \mathcal{S} = (\mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O}) \) be any CSTN, and \( s \) any scenario over \( \mathcal{P} \). The projection of \( \mathcal{S} \) onto \( s \) is noted \( \mathcal{S}(s) \)—the STN, \( (\mathcal{T}, \mathcal{C}_s^+) \), where:

\[
\mathcal{C}_s^+ = \{ (Y - X \leq \delta) \mid \exists \ell, (Y - X \leq \delta, \ell) \in \mathcal{C} \land s(\ell) = \top \}
\]

The truth values of propositions in a CSTN are not known in advance, but a π-dynamic execution strategy can react instantaneously to observations. To make instantaneous reactivity plausible, a π-execution strategy must specify an order of dependence among simultaneous observations.

Definition 9 (Order of dependence). For any scenario \( s \), and ordering \( (P_1, \ldots, P_k) \) of observation time-points, where \( k = |\mathcal{OT}| \), an order of dependence is a permutation \( \pi \) over \( \{1, 2, \ldots, k\} \); and for each \( P_i \in \mathcal{OT}, \pi(P_i) \in \{1, 2, \ldots, k\} \), the order of dependence \( \pi \) is satisfied in that order. For any non-observation time-point \( X \), we set \( \pi(X) = \infty \). Finally, \( \Pi_k \) denotes the set of all permutations over \( \{1, 2, \ldots, k\} \).

Definition 10 (π-Execution Strategy). Given any CSTN \( \mathcal{S} = (\mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O}) \), let \( k = |\mathcal{OT}| \). A π-execution strategy for \( \mathcal{S} \) is a mapping, \( \sigma : \mathcal{T} \rightarrow (\Psi \times \Pi_k) \), such that for each scenario \( s \), \( \sigma(s) \) is a pair \( (\psi, \pi) \) where \( \psi : \mathcal{T} \rightarrow \mathbb{R} \) is a schedule and \( \pi \in \Pi_k \) is an order of dependence. For any \( X \in \mathcal{T} \), \( [\sigma(s)]_X \) denotes the execution time of \( X \) (i.e., \( \psi(X) \)); and for any \( P_i \in \mathcal{OT}, \sigma(s)_P^\pi \), the position of \( P_i \) in the order of dependence (i.e., \( \pi(P_i) \)). Finally, a π-dynamic strategy must be coherent:

- for any scenario \( s \) and any \( P_i, Q_j \in \mathcal{OT} \), \( [\sigma(s)]_{P_i} < [\sigma(s)]_{Q_j} \) implies \( \pi(P_i) < \pi(Q_j) \), (i.e., if \( \sigma \) schedules \( P_i \) before \( Q_j \), then it orders \( P_i \) before \( Q_j \)).

Definition 11 (Viability). The π-execution strategy \( \sigma \) is called viable for the CSTN \( \mathcal{S} \) if for each scenario \( s \), the schedule \( \psi \) is a solution to the projection \( \mathcal{S}(s) \), where \( \sigma(s) = (\psi, \pi) \).

Definition 12 (π-History). Let \( \tau \) be any π-execution strategy for some CSTN \( \mathcal{S} = (\mathcal{T}, \mathcal{P}, \mathcal{C}, \mathcal{OT}, \mathcal{O}) \), \( s \) a scenario, \( t \) any real number, and \( d \in \{1, 2, \ldots, |\mathcal{OT}| \} \) any integer position (or infinity). The π-history of \((t, d)\) for the scenario \( s \) and strategy \( \sigma \)—denoted by \( \pi\text{Hist}(t, d, s, \sigma) \)—is the set:

\[
\{(p, s(p)) \mid P_i \in \mathcal{OT}, [\sigma(s)]_{P_i} \leq t, \pi(P_i) < d\}
\]

The π-history of \((t, d)\) specifies the truth value of each \( p \in \mathcal{P} \) that is observed before \( t \), or at \( t \) if the corresponding \( P_i \) is ordered before position \( d \) by the permutation \( \pi \).

Definition 13 (π-Dynamic Strategy). A π-execution strategy \( \sigma \), for a CSTN \( \mathcal{S} \) is called π-dynamic if for every pair of scenarios, \( s_1 \) and \( s_2 \), and every time-point \( X \in \mathcal{T} \):

let: \( t = [\sigma(s_1)]_X \), and \( d = [\sigma(s_1)]_X \).

if: \( \pi\text{Hist}(t, d, s_1, \sigma) = \pi\text{Hist}(t, d, s_2, \sigma) \)

then: \( [\sigma(s_2)]_X = t \) and \( [\sigma(s_2)]_X = d \).

Thus, if \( \sigma \) executes \( X \) at time \( t \) and position \( d \) in scenario \( s_1 \), and the histories, \( \pi\text{Hist}(t, d, s_1, \sigma) \) and \( \pi\text{Hist}(t, d, s_2, \sigma) \), are the same, then \( \sigma \) must also execute \( X \) at time \( t \) and in position \( d \) in \( s_2 \). (\( X \) may be an observation time-point.)

Definition 14 (π-Dynamic Consistency). A CSTN, \( \mathcal{S} \), is π-dynamically consistent (π-DC) if there exists a π-execution strategy for \( \mathcal{S} \) that is both viable and π-dynamic.
References


