

 Open access • Book Chapter • DOI:10.1007/978-3-319-07046-9_33

Dynamic Controllability and Dispatchability Relationships — [Source link](#)





Paul Morris

Published on: 19 May 2014 - Integration of AI and OR Techniques in Constraint Programming

Topics: Controllability

Related papers:

- [Temporal constraint networks](#)
- [Dynamic control of plans with temporal uncertainty](#)
- [A structural characterization of temporal dynamic controllability](#)
- [Handling contingency in temporal constraint networks: from consistency to controllabilities](#)
- [A fast incremental algorithm for maintaining dispatchability of partially controllable plans](#)

Share this paper:    

View more about this paper here: <https://typeset.io/papers/dynamic-controllability-and-dispatchability-relationships-12hokr8fda>

Dynamic Controllability and Dispatchability Relationships

Paul Morris

NASA Ames Research Center
Moffett Field, CA 94035, U.S.A.

Abstract. An important issue for temporal planners is the ability to handle temporal uncertainty. Recent papers have addressed the question of how to tell whether a temporal network is *Dynamically Controllable*, i.e., whether the temporal requirements are feasible in the light of uncertain durations of some processes. We present a fast algorithm for Dynamic Controllability. We also note a correspondence between the reduction steps in the algorithm and the operations involved in converting the projections to dispatchable form. This has implications for the complexity for sparse networks.

1 Introduction

Many Constraint-Based Planning systems (e.g. [1]) use Simple Temporal Networks (STNs) to test the consistency of partial plans encountered during the search process. These systems produce *flexible* plans where every solution to the final Simple Temporal Network provides an acceptable schedule. The flexibility is useful because it provides scope to respond to unanticipated contingencies during execution, for example where some activity takes longer than expected. However, since the uncertainty is not modeled, there is no guarantee that the flexibility will be sufficient to manage a particular contingency.

Many applications, however, involve a specific type of temporal uncertainty where the duration of certain processes or the timing of exogenous events is not under the control of the agent using the plan. In these cases, the values for the variables that are under the agent's control may need to be chosen so that they do not constrain uncontrollable events whose outcomes are still in the future. This is the *controllability* problem. By formalizing this notion of temporal uncertainty, it is possible to provide guarantees about the sufficiency of the flexibility.

In [2], several notions of controllability are defined, including *Dynamic Controllability* (DC). Roughly speaking, a network is dynamically controllable if there is an execution strategy that satisfies the constraints and depends only on knowing the outcomes of uncontrollable events up to the present time.

The fastest known algorithm for computing Dynamic Controllability (DC) is the $O(N^4)$ algorithm of [3] (N is number of nodes). That paper introduces a structural characterization of DC in terms of the absence of a particular type of cycle, called a *semi-reducible negative cycle*. This is analogous to the result

characterizing consistency of ordinary STNs in terms of the absence of negative cycles in the distance graph. Other properties of semi-reducible negative cycles have been studied by Hunsberger [4], who corrected a flaw in the formal definition of DC. An excellent tutorial on Dynamic Controllability is available online [5].

In this paper, we exploit recursive structure within the semi-reducible paths and present a new algorithm that runs in $O(N^3)$ time. We also consider the relationship to the dispatchability of the projections.

Other authors (e.g. [6, 7]) have pursued incremental algorithms where the Dynamic Controllability property is rechecked after the addition of a new edge to a network that has already been shown to be Dynamically Controllable. This corresponds to a common situation in temporal planning where edges are added incrementally to resolve flaws in the plan. These algorithms have been shown empirically to have $O(N^3)$ complexity for each increment on a suite of randomly generated problems. We do not address incrementality in this paper. Additional work has studied related concepts in wider contexts, e.g. [8, 9].

2 Background

This background section defines the basic formalism of Dynamic Controllability, following [3, 4].

A Simple Temporal Network (STN) [10] is a graph in which the edges are annotated with upper and lower numerical bounds. The nodes in the graph represent temporal events or *timepoints*, while the edges correspond to constraints on the durations between the events. Each STN is associated with a *distance graph* derived from the upper and lower bound constraints. An STN is consistent if and only if the distance graph does not contain a negative cycle. This can be determined by a single-source shortest path propagation such as in the Bellman-Ford algorithm [11] (faster than Floyd-Warshall for sparse graphs, which are common in practical problems). To avoid confusion with edges in the distance graph, we will refer to edges in the STN as *links*.

A Simple Temporal Network With Uncertainty (STNU) is similar to an STN except the links are divided into two classes, *requirement links* and *contingent links*. Requirement links are temporal constraints that the agent must satisfy, like the links in an ordinary STN. Contingent links may be thought of as representing causal processes of uncertain duration, or periods from a reference time to exogenous events; their finish timepoints, called *contingent timepoints*, are controlled by Nature, subject to the limits imposed by the bounds on the contingent links. All other timepoints, called *executable timepoints*, are controlled by the agent, whose goal is to satisfy the bounds on the requirement links. We assume the durations of contingent links vary independently, so a control procedure must consider every combination of such durations. Each contingent link is required to have non-negative (finite) upper and lower bounds, with the lower bound strictly less than the upper. We assume contingent links do not share finish points. (Networks with coincident contingent finishing points cannot be Dynamically Controllable.)

Choosing one of the allowed durations for each contingent link may be thought of as reducing the STNU to an ordinary STN. Thus, an STNU determines a family of STNs corresponding to the different allowed durations; these are called *projections* of the STNU.

Given an STNU with N as the set of nodes, a *schedule* T is a mapping

$$T : N \rightarrow \mathbb{R}$$

where $T(x)$ is called the *time* of timepoint x . A schedule is said to be *consistent* if it satisfies all the link constraints.

The *history* of a specific time t with respect to a schedule T , denoted by $T\prec t$, specifies the durations of all contingent links that have finished up to and including time t .

Hunsberger [4] corrected a flaw in the original definition of Dynamic Controllability by defining history in terms of a specific time rather than a timepoint; we follow that approach here and in the definition of dynamic strategy below. However, we also follow the variation of including the present that was introduced in [3]. The latter issue is whether the agent can react instantaneously to an observation to execute a new timepoint, or requires an infinitesimal amount of time to react. Both of these are mathematical idealizations: a realistic reaction might take a finite amount of time, which could be modeled by a separate link or folded into the contingent process being observed. The instantaneous idealization choice leads to a cleaner mathematical formulation and simpler algorithms.

An *execution strategy* S is a mapping

$$S : \mathcal{P} \rightarrow \mathcal{T}$$

where \mathcal{P} is the set of projections and \mathcal{T} is the set of schedules. An execution strategy S is *viable* if $S(p)$, henceforth written S_p , is consistent with p for each projection p .

An STNU is *Dynamically Controllable* if there is a *dynamic* execution strategy, that is, a viable execution strategy S such that

$$S_{p1}\prec t = S_{p2}\prec t \Rightarrow S_{p1}(x) = S_{p2}(x)$$

for each executable timepoint x and projections $p1$ and $p2$, where $t = S_{p1}(x)$ [4]. Thus, a Dynamic execution strategy assigns a time to each executable timepoint that may depend on the outcomes of contingent links in the past (or present), but not on those in the future. This corresponds to requiring that only information available from observation may be used in determining the schedule. We will use *dynamic strategy* in the following for a (viable) Dynamic execution strategy.

3 Previous Algorithms

In [12], an algorithm for Dynamic Controllability was presented that runs in pseudo-polynomial time. The algorithm analyzes triangles of links and possibly

tightens some constraints in a way that makes explicit the limitations to the execution strategies due to the presence of contingent links.

Some of the tightenings involve a novel temporal constraint called a *wait*. Given a contingent link AB and another link AC, the $\langle B, t \rangle$ annotation on AC indicates that execution of the timepoint C is not allowed to proceed until after either B has occurred or t units of time have elapsed since A occurred. More precisely, it corresponds to the constraint $C - A \geq \min(B - A, t)$. Thus, a wait is a ternary constraint involving A, B, and C. Note that a wait reduces to a binary constraint in any projection, since there the value $B - A$ is fixed.

The tightenings in the original algorithm, called *reductions*, were expressed in terms of rules that were applied to the STNU graph. We now review developments in [13, 3, 4], which re-express the reductions in a more mathematically concise form, using a derived graph.

An ordinary STN has an alternative representation as a *distance graph* [10]. Similarly, there is an analogous representation for an STNU called the *labeled distance graph* [13]. This is actually a multigraph (which allows multiple edges between two nodes), but we refer to it as a graph for simplicity. In the labeled distance graph, each requirement link $A \xrightarrow{[x,y]} B$ is replaced by two edges $A \xrightarrow{y} B$ and $A \xleftarrow{-x} B$, just as in an STN. For a contingent link $A \xrightarrow{[x,y]} B$, we have the same two edges $A \xrightarrow{y} B$ and $A \xleftarrow{-x} B$, but we also have two additional edges of the form $A \xrightarrow{b:x} B$ and $A \xleftarrow{B:-y} B$. These are called *labeled edges* because of the additional “b:” and “B:” annotations indicating the contingent timepoint B with which they are associated. Note especially the reversal in the roles of x and y in the labeled edges. We refer to $A \xleftarrow{B:-y} B$ and $A \xrightarrow{b:x} B$ as *upper-case* and *lower-case* edges, respectively. Observe that the upper-case labeled weight $B:-y$ gives the value the edge would have in a projection where the contingent link takes on its maximum value, whereas the lower-case labeled weight corresponds to the contingent link minimum value.

There is also a representation for a $A \xrightarrow{\langle B, t \rangle} C$ wait constraint in the labeled distance graph. This corresponds to a single edge $A \xleftarrow{B:-t} C$. Note the analogy to a lower bound. This weight is consistent with the lower bound that would occur in a projection where the contingent link has its maximum value.

We can now represent the tightenings in terms of the labeled distance graph. As in [3], we present a version of the rules that assumes the agent can react instantaneously to observed events.

The reductions from the classic algorithm are replaced by what is essentially a single reduction with different flavors, together with a label removal rule:

(UPPER-CASE REDUCTION)

$$A \xleftarrow{B:x} C \xleftarrow{y} D \quad \text{adds} \quad A \xleftarrow{B:(x+y)} D$$

(LOWER-CASE REDUCTION) If $x < 0$,

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

(CROSS-CASE REDUCTION) If $x < 0$, $B \neq C$,

$$A \xleftarrow{B:x} C \xleftarrow{c:y} D \quad \text{adds} \quad A \xleftarrow{B:(x+y)} D$$

(NO-CASE REDUCTION)

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

(LABEL REMOVAL REDUCTION) If $z \geq -x$,

$$B \xleftarrow{b:x} A \xleftarrow{B:z} C \quad \text{adds} \quad A \xleftarrow{z} C$$

With this reformulation, the “Case” (first four) reductions can all be seen as forms of composition of edges, with the labels being used to modulate when those compositions are allowed to occur. In light of this, the *unlabeled distance*¹ of a path in the labeled distance graph is defined to be the sum of edge weights in the path, ignoring any labels. Thus, the reductions preserve the unlabeled distance.

Morris [3] observes that a duration-uncertain contingent link $A \xrightarrow{[x,y]} B$ can be decomposed $A \xrightarrow{[x,x]} C \xrightarrow{[0,y-x]} B$ into a duration-certain part $A \xrightarrow{[x,x]} C$ and a “pure” duration-uncertain part $C \xrightarrow{[0,y-x]} B$. If this is done for all contingent links, the STNU is said to be in *Normal Form*. In that case, the contingent link lower bounds all become zero, so the LABEL REMOVAL reduction assumes a simpler form as follows.

(LABEL REMOVAL) If $x \geq 0$,

$$A \xleftarrow{B:x} C \quad \text{adds} \quad A \xleftarrow{x} C$$

We will assume in the remainder of the paper that STNU networks are in Normal Form since this simplifies the analysis and algorithms without loss of generality.

3.1 Path Transformations

The key to speeding up the determination of Dynamic Controllability is to perform the reductions in an organized way. This in turn is facilitated by considering the relationship of paths to Dynamic Controllability. To this end, Morris [3] defines a concept of *semi-reducible path*, which we review here.

¹ Terminology from [4]. Called *reduced distance* in [3], which is somewhat misleading.

An ordinary STN is consistent if and only if its distance graph does not contain a negative cycle. There is a related characterization of DC in terms of negative cycles in the labeled distance graph. This involves a notion of path transformation.

Consider a path \mathcal{P} that contains a subpath \mathcal{Q} between two points A and B and suppose \mathcal{Q} matches the left side of a reduction. Then applying the reduction to \mathcal{Q} yields a new edge e between A to B. Consider the path \mathcal{P}' obtained from \mathcal{P} by replacing \mathcal{Q} by e . We may regard \mathcal{P} as being *transformed* into \mathcal{P}' by the reduction. Note that \mathcal{P}' has the same unlabeled distance as \mathcal{P} since the reductions preserve unlabeled distance.

Definition 1. *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.*

The property of semi-reducible can be directly characterized in structural terms. The following notation is useful. We write $e < e'$ in \mathcal{P} if e is an earlier edge than e' in \mathcal{P} , where \mathcal{P} is a path in the labeled distance graph. If A and B are nodes in the path, we write $\mathcal{D}_{\mathcal{P}}(A, B)$ for the unlabeled distance from A to B in \mathcal{P} . We denote the start and end nodes of an edge e by $\text{start}(e)$ and $\text{end}(e)$, respectively.

Definition 2. *Suppose e is a lower-case edge in \mathcal{P} and e' is some other edge such that $e < e'$ in \mathcal{P} . The edge e' is a **drop edge** for e in \mathcal{P} if $\mathcal{D}_{\mathcal{P}}(\text{end}(e), \text{end}(e')) < 0$. The edge e' is a **moat edge** for e in \mathcal{P} if it is a drop edge and there is no other drop edge e'' such that $e'' < e'$ in \mathcal{P} . In this case, we call the subpath of \mathcal{P} from $\text{end}(e)$ to $\text{end}(e')$ the **extension** of e in \mathcal{P} . We say the moat edge is **unusable** if e and e' have labels that come from the same contingent link; otherwise it is **usable**.*

Thus, a drop edge is where the path following e becomes negative, and a moat edge is a closest drop edge. An unusable moat edge will have a label that is the upper-case version of the label on the lower-case edge.²

The extension subpath \mathcal{P} turns out to have a very useful property called the *prefix/postfix property*, which says that every nonempty proper prefix of \mathcal{P} has non-negative unlabeled distance and every nonempty proper postfix of \mathcal{P} has negative unlabeled distance. The *Nesting Lemma* [3] says if two prefix/postfix paths have a non-empty intersection, then one of the paths is contained in the other. This means that if a path has two subpaths with the property, then the subpaths are either nested or disjoint.

The main results of [3] provided a characterization of Dynamic Controllability in terms of the structure of the labeled distance graph, as follows.

Theorem 1. *A path \mathcal{P} is semi-reducible if and only if every lower-case edge in \mathcal{P} has a usable moat edge in \mathcal{P} .*

² We will see later the reductions may be viewed as performing composition of edges in projections, and these edges belong to incompatible projections.

Theorem 2. *An STNU is Dynamically Controllable if and only if it does not have a semi-reducible negative cycle.*

The labeled distance graph can be used to calculate distances between nodes in a similar manner to an ordinary STN distance graph, provided the restrictions imposed by the labels are respected. The approach in [3] calculates distances forward from each contingent timepoint looking for a usable moat edge to reduce away the associated lower-case edge. For the innermost nested extensions, this can be done in a single pass. This produces new edges that bypass these extensions, which decrements the level of nesting. Each pass has a complexity bound of $O(N^3)$ for the distance calculation. It was shown that the depth of nesting can be linearly bounded leading to a linear cutoff and an overall $O(N^4)$ complexity.

4 Cubic Algorithm

We now present a cubic algorithm for Dynamic Controllability using the same formal framework as [3], but with a different organization of the computation. Note that a moat edge must have negative unlabeled distance; thus, it must be either a negative ordinary edge, or a negative upper case labeled edge. Before leaving NASA, Nicola Muscettola [14, Personal Communication] proposed the following key ideas, which he anticipated would lead to a cubic algorithm. However, to the best of our knowledge, such an algorithm has not been published. We have formulated one based on these ideas and include it here. The key ideas are:

- Calculate distance *backwards* from potential moat edges. (That is, calculate distance *to* rather than distance *from*.)
- Calculate the distance over non-negative edges using Dijkstra’s algorithm [11].
- If a new negative edge is encountered, invoke a recursive call before continuing the distance calculation.
- A recursive cycle indicates the network is not Dynamically Controllable.

The backward distance calculation implicitly uses the reduction rules and may add new non-negative edges. The following example illustrates the approach (parentheses added for readability):

$$[(E \xrightarrow{4} B \xrightarrow{B:-2} A) \xrightarrow{b:0} B \xrightarrow{1} D \xrightarrow{D:-3} C] \xrightarrow{d:0} (D \xrightarrow{3} B \xrightarrow{B:-2} A) \xrightarrow{b:0} B \xrightarrow{-2} E$$

Consider a backward distance calculation starting at E. This will invoke a recursive call when it gets to A, which will add a $D \xrightarrow{1} A$ edge. (It also adds a $E \xrightarrow{2} A$ edge, which we ignore for now.) The top-level call will then continue using the $D \xrightarrow{1} A$ edge until it gets to C, where it causes another recursive call. When the call at C gets to A, it will use the already added $E \xrightarrow{2} A$ edge and leave behind a new $E \xrightarrow{0} C$ edge, at which point the top-level call resumes and encounters a recursive cycle.

An issue of special note is that Dijkstra’s algorithm is normally restricted to graphs with non-negative edges whereas in our case the initial edges connected to the Dijkstra source may be negative. However, it is easy to see (e.g., discussion in [15]) that the algorithm is still valid provided that (1) the only negative edges are the initial edges and (2) the propagation does not compute a negative distance to the source.³ If the propagation computes a negative distance to the source, this will be detected as a recursive cycle.

The goal of the computation is to discover semi-reducible negative cycles. It turns out that a restricted form of the reduction rules is sufficient to make this discovery because of the negativity. In particular, the Case reductions can all be restricted to $x < 0$ and $y \geq 0$. Application of the rules will stop when all the edges have the same sign, which must be negative since the whole cycle has negative unlabeled distance. (A rule cannot fail because of the label restrictions; if it did, the original cycle would contain an unusable moat edge and would not be semi-reducible.) A semi-reducible negative cycle is thus transformed by the rules to a cycle of all-negative edges, similar to a result of [7].

The restricted reduction rules are equivalent to the BackPropagate-Tighten rules used in the Incremental Dynamic Controllability work (e.g. [6, 7]).⁴ We will refer to the restricted reduction rules as *Plus-Minus* reductions since they involve a non-negative edge followed by a negative edge. We will also regard LABEL REMOVAL as being implicitly applied wherever it is applicable.

Before presenting the detailed algorithm (Fig. 1), we make a modification to the STNU to simplify the exposition. (An implementation could behave as if this modification is made without actually changing the data structures.) The modification separates the start nodes of contingent links from other contingent links and from the targets of ordinary negative edges. Thus,

$$\begin{aligned} B \Leftarrow A \Rightarrow C & \text{ becomes } B \Leftarrow A \xrightarrow{[0,0]} A' \Rightarrow C \\ B \Rightarrow A \Rightarrow C & \text{ becomes } B \Rightarrow A \xrightarrow{[0,0]} A' \Rightarrow C \\ B \xrightarrow{-x} A \Rightarrow C & \text{ becomes } B \xrightarrow{-x} A \xrightarrow{[0,0]} A' \Rightarrow C \end{aligned}$$

It is easy to see that the resulting STNU is equivalent to the original one. (Recall that we allow instantaneous reactions.) The modification keeps distance calculations involving different (and no) labels separate and adds at most $O(K)$ nodes and edges, where K is the number of contingent links.

The algorithm is summarized in Fig. 1. We have used indentation instead of begin-end to set off blocks of code. The `continue` statement, as in Java, skips

³ Consider the proof of correctness [11] of the usual algorithm. This relies on the fact that the distance to head nodes of the priority queue cannot be superseded by paths from later nodes, which start at a greater distance and are over non-negative edges. In our case, after processing the initial node, this will still be true for subsequent head nodes because paths from later nodes will use non-initial edges.

⁴ They have more rules because of multiple choices of focus edge, and because they make a distinction between direct and derived upper-case edges.

```

Boolean procedure determineDC()
  for each negative node n do
    if DCbackprop(n) = false
      return false;
    return true;
  end procedure

Boolean procedure DCbackprop(source)
00 if ancestor call with same source
01   return false;
02 if prior terminated call with source
03   return true;
04 distance(source) = 0;
05 for each node x other than source do
06   distance(x) = infinity;
07 PriorityQueue queue = empty;
08 for each e1 in InEdges(source) do
09   Node n1 = start(e1);
10   distance(n1) = weight(e1);
11   insert n1 in queue;
12 while queue not empty do
13   pop Node u from queue;
14   if distance(u) >= 0
15     Edge e' = new Edge(u, source);
16     weight(e') = distance(u);
17     add e' to graph;
18     continue;
19   if u is negative node
20     if DCbackprop(u) = false
21       return false;
22   for each e in InEdges(u) do
23     if weight(e) < 0
24       continue;
25     if e is unsuitable
26       continue;
27     Node v = start(e);
28     new = distance(u) + weight(e);
29     if new < distance(v)
30       distance(v) = new;
35     insert v into queue;
36 return true;
end procedure

```

Fig. 1. Cubic Algorithm

to the next turn of the immediately enclosing loop. A negative node is a node that is the target of some negative edge. There is a separate distance function for the distance to each source, but we have abbreviated $\text{distance}(x, \text{source})$ as $\text{distance}(x)$ to avoid clutter.

This is essentially a distance-limited version of Dijkstra's algorithm except for lines 00-01 and 19-21, which deal with the recursive aspect, lines 02-03, which short-circuit later calls with the same source, lines 08-11, which unroll the initial propagation, lines 14-18, which add non-negative edges to the graph, and the unsuitability condition in lines 25-26, which occurs if the source edge is unusable for e (from the same contingent link).⁵ The distance limitation occurs at the first non-negative value reached along a path (where a non-negative edge is added).

Notice that if the e in line 22 is a non-negative suitable edge, then since $\text{distance}(u)$ is negative, the Plus-Minus reductions will apply. The derived distance in line 28 will be that of either an ordinary or an upper-case edge. The added e' edge in line 17 is ordinary (by virtue of LABEL REMOVAL if necessary).

The whole algorithm terminates if the same source node is repeated in the recursion; thus, an infinite recursion is prevented. We will show that this condition occurs if and only if the STNU has a semi-reducible negative cycle. Thus, the algorithm does not require a subsidiary consistency check.

The early termination conditions in lines 00-03, which prevent infinite recursion and multiple calls with the same source, can be detected by marking schemes. Thus, the algorithm involves at most N (number of nodes in the network) non-trivial calls to DCbackprop, each of which (not including the recursive call) has complexity $O(N^2)$ if a Fibonacci heap is used for the priority queue, giving $O(N^3)$ in all. At most $O(N^2)$ edges are added to the graph; this cost is absorbed by the $O(N^3)$ overall complexity. The early termination calls to DCbackprop have $O(1)$ cost and come from superior calls or from determineDC. The former may be absorbed into the cost of line 20, while there are at most N of the latter. We now turn to the task of proving correctness.

Theorem 3. *The DCbackprop procedure encounters a recursive repetition if and only if the STNU is not Dynamically Controllable.*

Proof. Suppose first there is a recursive repetition. Note that if DCbackprop calls itself recursively, then there is a (backwards) negative path from the source of the superior call to that of the inferior. Since the distance calculations involve applications of the Plus-Minus reductions, that implies there is a reducible negative path from the first source to the second. Thus, a recursive repetition involves a cycle stitched together from reducible negative paths, which is a semi-reducible negative cycle.

Suppose conversely that the STNU is not Dynamically Controllable. Then it must have some semi-reducible negative cycle \mathcal{C} . The intuition behind the proof is that the negative segments in \mathcal{C} will either be bypassed, or will pile

⁵ It is useful to think of the distance calculation as taking place in the projection where any initial contingent link takes on its maximum duration and every other contingent link has its minimum. An unsuitable edge does not belong to that projection.

up against each other. Since they cannot all be bypassed, this will result in a recursive cycle. For the argument, it is convenient to temporarily remove lines 00-01 of the algorithm. In that case, a recursive cycle will result in an infinite recursion rather than returning **false** and we can talk about termination instead of what value is returned.

Note that for every negative node A in \mathcal{C} , $\text{DCbackprop}(A)$ will be called eventually, either as a recursive call or as a top-level call from determineDC . By line 17, the execution of $\text{DCbackprop}(A)$ may add a non-negative edge BA from some other node B in \mathcal{C} to A . We will call this a *cross-edge*. Since the Dijkstra algorithm computes shortest paths, we have $\text{weight}(BA) \leq \mathcal{D}_{\mathcal{C}}(B, A)$.

If there is no infinite recursion, then every call to DCbackprop must terminate. Our strategy will be to show that every terminating call to $\text{DCbackprop}(A)$ for some A in \mathcal{C} will add at least one cross-edge. These will then bypass all the negative edges in \mathcal{C} , which contradicts the fact that \mathcal{C} is a negative cycle.

First assume all the non-negative edges in \mathcal{C} are ordinary edges. (Lower-case edges add a slight complication that we will address in due course.) Consider the very first termination of a $\text{DCbackprop}(A)$ call for A in \mathcal{C} . Note that the execution cannot have included a recursive call; otherwise the recursive call would have terminated first. The backward Dijkstra propagation must reach the predecessor A' of A in \mathcal{C} . This cannot be a negative node, since otherwise it would cause a recursive call. If the distance to A' is non-negative, then $\text{DCbackprop}(A)$ will add a non-negative $A'A$ edge. Otherwise the propagation must reach predecessor A'' of A' since A' is not a negative node. (Thus, $A'A$ is a non-negative edge.) The propagation will continue to predecessors until a non-negative distance is reached. This must happen eventually. (Otherwise the propagation would continue all the way back to A and cause a recursive loop.) Then the execution of $\text{DCbackprop}(A)$ will add a cross-edge before it terminates. For the inductive step, the argument is similar, except any recursive calls that have already terminated will have left behind cross-edges, and the propagation will be over those rather than the edges in the original cycle.

Now consider the case where \mathcal{C} contains lower-case edges. The difficulty here is that subpaths of \mathcal{C} are not necessarily shortest. Since the cross-edges resulting from the Dijkstra calculation are shortest paths, the shortenings could result in a closer moat edge for a lower-case edge. However, by Theorem 3 of [3], we can assume without loss of generality that \mathcal{C} is *breach-free*. (A lower-case edge in the cycle has a *breach* if its extension contains an upper-case edge from the same contingent link.) In that case, the closer moat edge would still be usable.

Thus, we have shown every terminating call to DCbackprop leaves behind a cross-edge, which results in a contradiction. It follows there is some non-terminating call, i.e., an infinite recursion. When we put back lines 00-01, the recursive cycle is trapped, and results instead in a determination that the STNU is not Dynamically Controllable. \square

The algorithm as presented only adds non-negative edges, which are the only ones needed for the Dijkstra calculations. However, derived negative edges are implicit in the distance calculation. Thus, when $\text{distance}(u)$ is negative in line 14

of the algorithm, we could infer and save a negative edge or wait condition. We will call the algorithm that does this `determineDC+`. Results in the next section show that the network resulting from `determineDC+` is suitable for execution.

5 Dispatchability

Previous papers (e.g. [6]) have noted a relationship between Dynamic Controllability and dispatchability, but have not investigated it formally. In this section, we clarify the relationship between dispatchability and Dynamic Controllability and explore how dispatchability applies to an STNU. We start by relating dispatchability of an STNU to the better-understood dispatchability of its STN projections. Since wait edges may be needed for this property to hold, we consider an *extended* STNU (ESTNU) that may include wait edges. Recall that wait edges reduce to ordinary edges in a projection.

A *dispatching execution* [16] is one that respects direct precedence constraints, and propagates execution times only to neighboring nodes, but otherwise is free to execute a timepoint at any time within its propagated bounds. An STN is *dispatchable* if a dispatching execution is guaranteed to succeed. It is shown in [16] that every consistent STN can be reformulated into an equivalent *minimum dispatchable network*. The reformulation procedure first constructs the AllPairs network and then eliminates *dominated* edges that are not needed for dispatchability. A fast version [17] of the algorithm computes distances from one node at a time and uses that to determine which edges from that node are dominated.

We can extend these notions to an ESTNU by essentially pretending that contingent timepoints are executed by the agent and propagating the observed time. For an ESTNU dispatching execution, we require the free choices are made only for executable timepoints, and respect the waits and precedences. We cannot directly mandate that the contingent timepoints respect the precedences; however (see proof of next result), this is indirectly achieved if the projections of the ESTNU are dispatchable. Note that such a strategy does not depend on future events. This leads to the following.

Definition 3. *An ESTNU is dispatchable if every projection is dispatchable.*

Theorem 4. *A dispatchable ESTNU is Dynamically Controllable.*

Proof. Suppose all the projections are dispatchable. We show that an ESTNU dispatching execution will satisfy precedence constraints for the contingent timepoints. Suppose otherwise and let $A \xrightarrow{[x,y]} B \xrightarrow{-z} C$ be the subnetwork where a precedence is violated for the first time in some projection. Consider the state of the execution after A and strictly before B, but within $z/2$ units of time prior to B. This is a dispatching execution in the STN sense since no precedence has been violated yet. However, the constraints in the projection now force C into

the past although it has not been executed yet, which implies the projection is not dispatchable [17] contrary to our assumption.⁶

Thus, the ESTNU dispatching strategy restricted to each projection is a dispatching execution. If it fails, the projection in which it fails, and hence the ESTNU, are not dispatchable. \square

Since dispatchability is itself a desirable property, this suggests the objective of transforming an STNU into an ESTNU such that each projection is dispatchable, preferably in minimum dispatchable form [16]. It is natural to consider if the fast algorithm discussed in [17], or some variant, can be adapted for this purpose. As it turns out, the `determinDC+` algorithm may itself be viewed as such a variant, although it does not achieve minimum form. First we prove some basic facts about dispatchability of an STN in preparation for considering dispatchability of STNU projections.

Recall that a path has the prefix/postfix property if every nonempty proper prefix is non-negative and every nonempty proper postfix is negative. It turns out that prefix/postfix paths in an STN are related to edges in a minimum dispatchable network (MDN) for the STN. By results in [16, 17], an MDN edge is either undominated or is one (which may be arbitrarily chosen) of a group of concurrent mutually dominating edges that are not strictly dominated.

Theorem 5. *Every consistent STN has a minimum dispatchable network such that if AB is an edge in that MDN, then there is a shortest path \mathcal{P} from A to B in the STN such that \mathcal{P} has the prefix/postfix property.*

Proof. First suppose AB is an undominated MDN edge. Let \mathcal{P} be any shortest path from A to B in the STN and let C be any node strictly between A and B in \mathcal{P} . Consider the case where AB is negative. Then the prefix AC must be non-negative; otherwise AB would be lower-dominated [16]. It follows that the postfix CB is negative. On the other hand, if AB is non-negative, then CB must be negative (otherwise AB would be upper-dominated), and then AC is non-negative.

For the mutual dominance case, we restrict the MDN edge choice. Among a group of mutually dominating edges that are not strictly dominated, we choose an MDN edge AB such that a shortest path \mathcal{P} from A to B does not contain a shortest path for another edge in the group. We can then use an argument similar to the undominated case because if AB is dominated by AC or CB , it would be strictly dominated. \square

This leads to a sufficient condition for dispatchability.

Definition 4. *An STN is prefix/postfix complete (PP complete) if whenever the distance graph has a shortest path from A to B with the prefix/postfix property, there is also a direct edge from A to B whose weight is equal to the shortest path distance.*

⁶ As an example, given an STNU $A \xrightarrow{[2,4]} B \xrightarrow{-1} C$, the minimum dispatchable network for the minimum-duration projection includes $A \xrightarrow{1} C$, which makes C precede B .

Theorem 6. *A consistent STN that is PP complete is dispatchable.*

Proof. By Theorem 5, the STN contains all the edges of one of its MDNs; thus, it is dispatchable. \square

Prefix/postfix paths have a well-behaved structure. Suppose a path \mathcal{P} has the prefix/postfix property. Clearly if it has more than one edge, then the first edge must be non-negative and the last edge must be negative. Now consider a negative edge e in the interior of \mathcal{P} . The proper prefix of \mathcal{P} that ends with e will be non-negative; thus, there must be some closest e' to e such that the subpath \mathcal{Q} from e' to e is non-negative. It is not difficult to see that \mathcal{Q} will also have the prefix/postfix property. We will call the subpath \mathcal{Q} the *train* of e in \mathcal{P} . We get a similar train for every negative edge within \mathcal{P} . Since they all have the prefix/postfix property, the trains must be nested or disjoint. (This result is similar to the Nesting Lemma for extensions of lower-case edges.) Note that an innermost nested train consists of a negative edge preceded by all non-negative edges, and (like all trains) has non-negative total distance.

For an ESTNU, it turns out that we only need to ensure PP completeness for a subset of the projections. The *AllMin* projection is where every contingent link takes on its minimum duration. In an *AllMinButOne* projection, one of the contingent links takes on its maximum duration, and every other contingent link takes on its minimum duration. We will call these the *basic projections* of an ESTNU.

Theorem 7. *Given an ESTNU, if the basic projections are PP complete, then every projection is PP complete.*

Proof. Suppose the basic projections are PP complete, and consider a prefix/postfix shortest path \mathcal{P} in one of the projections. The proof is by induction on the depth of nesting of the trains in \mathcal{P} . Recall that an innermost train must consist of a negative edge preceded by all non-negative edges. It is not hard to see that such a path has its minimum distance in one of the basic projections. The subpath corresponding to the train will still have the prefix/postfix property there. (The postfixes are not larger, and the proper prefixes have non-negative edges.) By PP-completeness of that basic projection, there must be an edge in the ESTNU that bypasses the subpath. Its distance cannot be less than the subpath since \mathcal{P} is a shortest path; thus, they have the same distance. This reduces the depth of nesting. If the depth is zero, the same reasoning can be used to infer a bypass edge for the whole path. \square

This gives us some insight into the functioning of the determineDC+ algorithm. The nested trains cause recursive calls of DCBackprop. If DCBackprop(A) is called where AB is a contingent link, then the algorithm may be viewed as adding PP-completeness edges for the AllMinButOne projection for AB. If instead, the call is where A is the target of ordinary negative edges, then it is adding PP-completeness edges for the AllMin projection. The recursive calls ensure that non-negative edges are added in the order corresponding to the nesting. In summary, the algorithm is extending the STNU so that it is prefix/postfix complete with respect to the basic projections, and thus is dispatchable.

6 Closing Remarks

Note that in contrast to the fast MDN algorithm [17], all the edges in the original network are kept by determineDC+. The algorithm may also add unneeded dominated edges in addition to the non-dominated ones that it needs. The number of added edges is significant because the complexity of a Dijkstra computation is sensitive to the number of edges. This suggests a potential avenue for future improvement.

A class of STNs is said to be *sparse* if the number of edges E is a fixed multiple of the number of nodes N (i.e., E scales as $O(N)$). The cost of one Dijkstra call is $O(E + N \log N)$, which is $O(N \log N)$ for a sparse network. Networks encountered in practical problems tend to be sparse. Typically for an STN, if the original network is sparse, the minimum dispatchable network is also sparse [17],⁷ since it essentially contains the same information in a concise form. It is reasonable to think the same might be true for an STNU if only non-dominated edges are added. Thus, if the algorithm could be improved to not add any dominated edges, then the complexity might in practice be comparable to that of the Fast Dispatchability algorithm, i.e., $O(N^2 \log N)$ for sparse networks. The issue essentially is to prune unneeded edge additions from each recursive call before the higher-level calls use them.

It might seem the ideal solution would be to adapt the fast minimum dispatchability algorithm [17] (FMDA) directly. For an STNU the distance calculation could be backwards and invoke recursive calls at negative nodes. However, the adaption would also need to handle, or sidestep, the contraction of rigid components in FMDA, which may be complicated by the fact that a contingent link is itself a rigid constraint in a projection. Besides that, there is the question of how to adapt the reweighting approach of FMDA (which makes possible a Dijkstra computation where the original weights may be negative), so that it works for all the basic projections. Also, to be worth it, the adaptations would need to fit within the existing FMDA cost. These are challenges for future research.

Acknowledgment

This paper owes a profound debt to the many insights of Nicola Muscettola, which include the key ideas underpinning the cubic algorithm. The current author is responsible for the formal treatment and proofs, and the results relating Dispatchability and Dynamic Controllability.

References

1. Muscettola, N., Nayak, P., Pell, B., Williams, B.: Remote agent: to boldly go where no AI system has gone before. *Artificial Intelligence* **103**(1-2) (August 1998) 5–48
2. Vidal, T., Fargier, H.: Handling contingency in temporal constraint networks: from consistency to controllabilities. *JETAI* **11** (1999) 23–45

⁷ Artificially constructed exceptions are unlikely to occur in practice.

3. Morris, P.: A structural characterization of temporal dynamic controllability. In: CP. (2006) 375–389
4. Hunsberger, L.: Magic loops in simple temporal networks with uncertainty - exploiting structure to speed up dynamic controllability checking. In: ICAART (2). (2013) 157–170
5. Hunsberger, L.: Tutorial on dynamic controllability. <http://icaps13.icaps-conference.org/wp-content/uploads/2013/06/hunsberger.pdf> (2013)
6. Shah, J.A., Stedl, J., Williams, B.C., Robertson, P.: A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In Boddy, M.S., Fox, M., Thibaux, S., eds.: ICAPS, AAAI (2007) 296–303
7. Nilsson, M., Kvarnström, J., Doherty, P.: Incremental Dynamic Controllability Revisited. In: Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS), AAAI Press (2013)
8. Rossi, F., Venable, K.B., Yorke-Smith, N.: Uncertainty in soft temporal constraint problems: A general framework and controllability algorithms for the fuzzy case. *Journal of Artificial Intelligence Research* **27** (December 2006) 617–674
9. Tsamardinos, I., Pollack, M.E.: Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence* **151** (2003) 43–89
10. Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. *Artificial Intelligence* **49** (May 1991) 61–95
11. Cormen, T., Leiserson, C., Rivest, R.: *Introduction to Algorithms*. MIT press, Cambridge, MA (1990)
12. Morris, P., Muscettola, N., Vidal, T.: Dynamic control of plans with temporal uncertainty. In: Proc. of IJCAI-01. (2001)
13. Morris, P., Muscettola, N.: Dynamic controllability revisited. In: Proc. of AAAI-05. (2005)
14. Muscettola, N.: (2006) Personal Communication.
15. Web. <http://stackoverflow.com/questions/3833500/dijkstras-algorithm-with-negative-edges-on-a-directed-graph> (2010)
16. Muscettola, N., Morris, P., Tsamardinos, I.: Reformulating temporal plans for efficient execution. In: Proc. of Sixth Int. Conf. on Principles of Knowledge Representation and Reasoning (KR'98). (1998)
17. Tsamardinos, I., Muscettola, N., Morris, P.: Fast transformation of temporal plans for efficient execution. In: Proc. of Fifteenth Nat. Conf. on Artificial Intelligence (AAAI-98). (1998)