A Fast Incremental Dynamic Controllability Algorithm

John Stedl and Brian Williams
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar St. Room 32-G275, Cambridge, MA 02139
stedl@mit.edu, williams@mit.edu

Proceedings of the ICAPS Workshop on Plan Execution, Monterey, CA, June 2005.

Abstract

In most real-world planning and scheduling problems, the plan will contain activities of uncertain duration whose precise timing is observed rather than controlled by the agent. In many cases, in order to satisfy the temporal constraints imposed by the plan, the agent must dynamically adapt the schedule of the plan in response to these uncertain observations. Previous work has introduced a polynomial-time *dynamic controllability (DC)* algorithm, which reformulates the temporal constraints of the plan into a *dispatchable* form, which is amenable for efficient dynamic execution.

In this paper we introduce a novel, fast Incremental DC algorithm (Fast-IDC) that (1) very efficiently maintains the dispatchability of a partially controllable plan when the subset of the constraints change and (2) efficiently reformulates previously unprocessed, partially controllable plans for dynamic execution. This new Fast-IDC algorithm has been implemented in C++ and shown to run in $O(N^3)$ time when reformulating unprocessed plans and in O(N) time when maintaining the dispatchability of the plan.

Introduction

In most real-world planning and scheduling problems, the timing of some of the events will be controlled by the agent; while others will be controlled by nature. For example, a Mars rover is capable of controlling when it starts driving to a rock; however, its precise arrival time is determined by environmental factors.

In order to be confident that the agent will successfully execute a plan that contains activities of uncertain duration, it is insufficient to merely guarantee that there exists a feasible schedule. Instead, the agent must ensure there is a strategy to consistently schedule the controllable events for all possible outcomes of the uncertain durations. The problem of determining if a viable execution strategy exists was first formally addressed by [Vidal 1996, Vidal and Fargier 1999]. This work has identified three primary levels of *controllability*: Strong, Dynamic and Weak. Controllability refers to the ability to "control" the consistency of the schedule, despite the uncertainty in the plan.

In this paper we are concerned with *dynamic controllability*, in which agent adapts the schedule of the plan based on the uncertain durations that are observed at execution time. Informally, a plan is dynamically controllable if there is a successful execution strategy that assigns execution times to the controllable events, which only depends on past outcomes and satisfies the timing constraints in the plan for all possible execution times of uncontrollable events. Furthermore, a plan is *dispatchable* if there is a means to efficiently schedule a dynamically controllable plan.

[Morris 2001] introduced a polynomial time dynamic controllability (DC1) algorithm to reformulate a partially controllable plan into a dispatchable plan. In this paper we improve upon this algorithm by introducing a fast incremental dynamic controllability algorithm (Fast-IDC). This Fast-IDC provides two key related capabilities. First it enables an agent to quickly maintain the dispatchability of the plan when only some of the constraints change. Second, we show how to efficiently apply this IDC algorithm in the startup case in order to reformulate unprocessed plans as fast or faster than DC1. This first capability becomes particularly important when dealing with highly agile systems, such as unmanned aerial vehicles, where there may not be enough time to restart the reformulation processed when some of the constraints change.

[Morris 2001] showed that converting a partially controllable plan into a dispatchable plan is reduced to repeatedly applying a set of constraint propagations. These constraint propagations introduce either simple temporal constraints or "wait" constraint [Morris 2001]. In this paper we show how exploit the structure of the plan in order to efficiently apply these constraint propagations.

Specifically, we introduce and exploit a property called *pseudo-dispatchability*, which enables an efficient, recursive constraint propagation scheme, called *dispatchability-back-propagation (DBP).* The sub-term "back-propagation" refers to the fact that the constraints only need to be propagated toward the start of the plan. DBP is efficient because (1) each constraint only need to be resolved with a subset of the constraints in the plan (2) it can operated on a trimmed plan in which redundant constraints are removed.

When reformulating unprocessed plans, we efficiently apply DBP by using two techniques. First we remove the redundant constraints before performing constraint propagation, which significantly reduces the number of propagations required. Second, we intelligently initiate the

DBPs such that the algorithm continuously reduces the size of the problem. Our purely recursive approach removes the need to perform repeated calls to an $O(N^3)$ All-Pairs Shortest-Path (APSP) algorithm, as required by the DC algorithm introduced by [Morris 2001].

First we review Simple Temporal Networks (STNs) Simple Temporal Networks with Uncertainty (STNUs). Then we describe how to perform DBP on STNs. Next we extend this DBP framework to STNUs. Then we introduce our Incremental DC (IDC) algorithm to handle the case when only one constraint changes. Next we show how to efficiently apply this IDC algorithm for both unprocessed plans and when multiple constraints change. Finally, we present some experimental results of our IDC algorithm.

Background

A Simple Temporal Network with Uncertainty [Vidal and Fargier 1999] is an extension of a STN [Dechter 1991] that distinguishes between controllable and uncontrollable events. A STNU is a directed graph, consisting of a set of nodes, representing *timepoints*, and a set of edges, called links, constraining the duration between the timepoints. The links fall into two categories: *contingent links* and *requirement links*. A contingent link models an uncontrollable process whose uncertain duration, ω, may last any duration between the specified lower and upper bounds. A requirement link simply specifies a constraint on the duration between two timepoints. All contingent links terminate on a *contingent timepoint* whose timing is controlled by nature. All other timepoints are called requirement timepoints and are controlled by the agent.

Definition (STNU [Vidal 1999]): *A STNU is a 5-tuple <N, E, l, u, C>, where N is a set of timepoints, E is a set of edges and* $l : E \rightarrow \Re \cup \{-\infty\}$ *and* $u : E \rightarrow \Re \cup \{+\infty\}$ *are functions mapping the edges to lower and upper bound temporal constraints. The STNU also contains C, which is a subset of the edges that specify the contingent links, the others being requirement links. We assume* $0 < l(e) < u(e)$ *for each contingent link*

To support efficient inference, a STNU is mapped to an equivalent distance graph [Dechter 1991], called a Distance Graph with Uncertainty (DGU), where each link of the STNU, containing both lower and upper bounds, is converted into a pair DGU edges, containing only an upper bound constraint. In the DGU, the distinction between a contingent and a requirement edge is maintained. For example consider the triangular STNU and associated DGU shown in Figure 1.

Similar to an STN, a STNU is consistent only if its associated DGU contains no negative cycles [Dechter 1991]. This can be efficiently checked by applying the Bellman-Ford SSSP algorithm [CLR 1990] on the DGU. However consistency does not imply dynamic controllability.

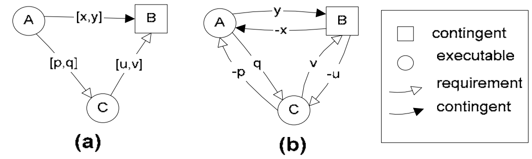

Figure 1 (a) Triangular STNU and (b) DGU

In order for the STNU to be dynamically controllable, each uncontrollable duration, ω_i, must be free to finish any time between $[l_i,u_i]$, as specified by the contingent link, C_i. The set of all implicit constraints contained in the STNU can be made explicit by computing the APSP-graph of the DGU via the Floyd-Warshall algorithm [CLR 1990].

If temporal constraints of the plan imply strictly tighter bounds on an uncontrollable duration, then that uncontrollable duration is *squeezed* [Morris 2001] and the plan is not dynamically controllable. In this case there exists a *situation* [Vidal 1999] where the outcome of the uncontrollable duration may result in an inconsistency. A STNU is *pseudo-controllable* [Morris 2001] if it is both temporally consistent and none of its uncontrollable durations are squeezed.

In this paper we are interested in preparing the STNU for dynamic execution in which a dispatcher [Morris 2001] uses the associated DGU to schedule timepoints at execution time. Even if a STNU is pseudo-controllable, the uncontrollable durations may be squeezed at execution time [Morris 2001].

The dynamic controllability (DC) reformulation algorithm introduced by [Morris 2001] adds additional constraints, simple temporal constraints and “wait” constraints, to the plan, in order to enable the dispatcher to consistently schedule the plan at execution time without squeezing the uncontrollable durations. In our Fast-IDC algorithm we apply these tightenings efficiently.

Incremental STN Dispatchability Maintenance

The speed of our Incremental Fast-DC algorithm depends on a technique called *dispatchability-back-propagation (DBP)*. In this section we introduce the DBP rules for STNs. In the next section we extend these rules for STNUs.

In order to address real-time scheduling issues, [Muscettola 1998] showed that any consistent STN can be converted into an equivalent *dispatchable* distance graph, which can be dynamically scheduled using a locally propagating dispatching algorithm [Muscettola 1998]. Furthermore, [Muscettola 1998] showed that the dispatcher can run efficiently if the redundant constraints are removed from the plan, forming a *minimal dispatchable graph.*

The dispatching algorithm schedules and executes the timepoints at the same time. The dispatcher works by maintaining a list of enabled timepoints along with a feasible execution widow, $W_x \in [lb_x,ub_x]$, for each timepoint X. When the dispatcher executes a timepoint, the dispatcher both updates the list of enabled timepoints and propagates this execution time to update the execution window of unexecuted timepoints. Specifically, when a timepoint A is executed, upper-bound updates are propagated via all outgoing, non-negative edges AB and lower-bound updates are propagated via all incoming negative edges, CA. The dispatching algorithm is free to schedule timepoint X anytime within X's execution window, as long X is enabled. A timepoint X is enabled if all timepoints that must precede X have been executed.

In order to develop the DBP rules for STN's we exploit the dispatchability of the plan. For a dispatchable graph, the dispatcher is able to guarantee that it can make a consistent assignment to all future timepoints, as long as each scheduling decision is consistent with the past. Therefore, in order to maintain the dispatchability of the plan when a constraint is modified, we only need to make sure that the change is consistent (resolved) with the past; the dispatcher will ensure that this constraint change is consistent (resolved) with the future at execution time.

Specifically, when an edge X changes, it only needs to be resolved with the set of edges that may cause an inconsistency with the time window update propagated by edge X at execution time. These set of edges are called *threats.*

We call the process of ensuring an edge change is consistent with the past, *Dispatchability Back-Propagation (DBP).*

Lemma (STN-DBP) Given a dispatchable STN with associated distance graph G,

(i) Consider any tightening (or addition) of an edge AB with d(AB) = y, where y>0 and A≠B; for all edges BC with d(BC)= u, where u <= 0, we can deduce a new constraint AC with d(AC) = y + u.

(ii) Consider any tightening (or addition) of an edge BA with d(BA)= x, where x <= 0 and A≠B; for all edges CB with d(CB)= v, where v >= 0, we can deduce a new constraint CA with d(CA) = x+v.

Proof: (i) During execution, a non-negative edge AB propagates an upper bound to B of $ub_B = T(A) + d(AB)$. A negative edge BC propagates a lower bound to B of $lb_B = T(C) - d(BC)$. At execution time, changing AB will be consistent if $ub_B >= lb_B$ for any C, or T(A) + d(AB) >= T(C) - d(BC), which implies T(A) - T(C) < d(AB) + d(BC). Adding an edge CA of d(AB) + d(BC) to G encodes this constraint. Similar reasoning applies for case (ii) when a negative edge changes.

Recursively applying rules (i) and (ii), when an edge changes in a dispatchable distance graph, will either expose a direct inconsistency or result in a dispatchable graph. This back-propagation technique only requires a subset of the edges to be resolved with the change, instead of all the edges, which would happen if we were to re-compute the APSP-graph every time an edge changed. Specifically

For example, consider the series of STN-DBPs required when the edge DC, shown in Figure 2a, is changed in the originally dispatchable graph. This change must be back-propagated through the threats, CB, and BD. The modified edges, BC and CC, resulting from this back-propagation, are shown in Figure 2-b. The self-loop CC is consistent and has no threats; however, the edge BC must be back-propagated through its threats, CB and CA. The results of this back-propagation modifies edges BB and BA, as shown in Figure 2-c. Now BA is threatened by AB. The

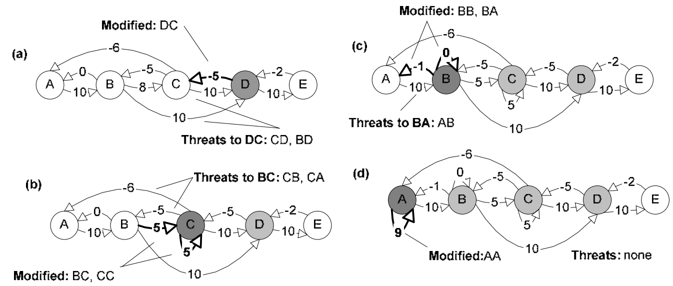

Figure 2 STN Back-Propagation Example

next round of back-propagation results in a dispatchable graph, as shown in Figure 2-d. In this example, only 5 propagations were required; recomputing the APSP-graph using the Floyd-Warshall APSP algorithm would have required 125 propagations.

In order to apply DBP to distance graphs with uncertainty (DGUs), we introduce the idea of *pseudo-dispatchability.* If we ignore the distinction between contingent and requirement edges in the DGU, then the DGU is effectively converted into distance graph (DG). If this associated DG is dispatchable, then we say the DGU *pseudo-dispatchable*. If a DGU is both pseudo-dispatchable and pseudo-controllable, then its dispatchability is only threatened by possible squeezing of the uncontrollable durations at execution time. In the next section we show how to exploit the pseudo-dispatchability of a plan in order to efficiently reformulate the plan in order to prevent this squeezing from occurring. Furthermore, the pseudo-dispatchability of the DGU is maintained by recursively applying the STN-DBP rules.

We also introduce the term *pseudo-minimal dispatchable graph* (PMDG), which is a DGU that is both pseudo-dispatchable and contains the fewest number of edges. The PMDG can be computed by applying either the "slow" STN reformulation algorithm introduced by [Muscettola 1998], or the "fast" STN reformulation algorithm introduced by [Tsamardinos 1998], to the DGU (ignoring the distinction between contingent and requirement edges).

Defining the DBP Rules for STNUs

In this section we unify reduction and regression rules introduced by [Morris 2001] with the STN Dispatchability Back Propagations (STN-DBP) rules described in the previous section, to form the DBP rules for STNUs. In the next section we use these rules to design our Fast Incremental DC (Fast-IDC) algorithm.

First we review the reduction rules introduced by [Morris 2001] which prevent the uncontrollable durations from being squeezed at execution time. Consider the triangular STNU and associated DGU shown in Figure 1. Assume that the STNU is both pseudo-controllable and in an APSP form.

Precede Case: u > 0: The precede reduction prevents the propagations from either CB or BC, from squeezing the contingent link AB.

Definition (Precede Reduction [Morris 2001]) *If u > 0, tighten AC to x-u, and edge CA to v-y.*

Unordered Case: v ≥ 0 and u ≤ 0: The unordered reduction prevents propagations through edge CB from squeezing the contingent link AB, when C executes first, yet allows B to propagate an upper bound through BC, when B executes first.

A conditional edge, introduced by [Morris 2001], must be added to the DGU in order to handle this case. This is slightly different from the "wait" constraint defined by [Morris 2001]; however, provides the same functionality. We call a DGU containing a set of conditional constraints, a Conditional Distance Graph with Uncertainty (CDGU). A conditional edge CA of <B,-t> specifies that A must wait at least *t* time units after A executes or until B executes, which ever is sooner. Note that the form of a conditional edge is similar to a negative requirement edge.

Definition (Unordered Reduction [Morris 2001]) *If v ≥ 0 and u ≤ 0, apply a conditional constraint CA of <B, v-y>.*

In some cases the conditional edge is unconditional. The unconditional unordered reduction describes when to convert the conditional edge into a requirement edge.

Definition (Unconditional Unordered Reduction [Morris 2001]) *Given a STNU with contingent link AB ∈ [x,y], and associated CDGU with a conditional constraint CA of <B,-t>, if x > t, then convert the conditional constraint CA into a requirement CA with distance –x.*

In order to prevent a conditional constraint from being violated at execution time, it must be regressed through the CDGU.

Lemma (Regression [Morris 2001]): *Given a conditional constraint CA of <B,t>, where -t is less than or equal to the upper bound of contingent link AB. Then (in a schedule resulting from a dynamic strategy):*

i.) If there is a requirement edge DC with distance w, where w ≥ 0 and D ≠ B, we can deduce a conditional constraint DA of <w+t, B>.

ii.) If t < 0 and if there is a contingent link DC with bounds [x,y] and B ≠ C, then we can deduce a conditional constraint DA of <x+t, B>.

Given that the plan is pseudo-controllable and pseudo-dispatchable, in order to maintain the dispatchability of the CDGU when a constraint changes, we only need iteratively apply all rules (STN-DBP, regression, and reductions) that pertain to that constraint. Table 1 summarizes the DBP rules used in our Fast-IDC algorithm. This unified set of rules enables each type of propagation to be interleaved. This differs from the technique used by [Morris 2001] that requires an APSP computation to perform the requirement edge propagations. In the next section we describe the process of iteratively applying the DBP for STNUs to create an incremental dynamic controllability algorithm.

Applying the DBP rules for STNUs

In this section we use the DBP rules in order to define an incremental algorithm for maintaining the dispatchability of a plan when one or more of the constraints change. In the next section we extend this algorithm to reformulate unprocessed plans into a dispatchable form.

The function BACK-PROPAGATE, shown in Figure 3, maintains the dispatchability of a CDGU, G, when an edge (u,v) changes. The function BACK-PROPAGATE recursively applies the DBP rules shown in Table 1, until either it detects a direct inconsistency or until no more propagations are required.

The BACK-PROPAGATE algorithm first checks if the edge (u,v) is a loop, (i.e. starts and ends on the same timepoint). If it is a positive loop, no more propagations are required and the algorithm returns true. If the edge is a negative loop, then an inconsistency is detected and the algorithm returns false.

Next the algorithm resolves all possible threats to (u,v) by applying the DBP rules in order to generate a candidate update edge (p,q). Two special conditions are considered if the candidate is a conditional edge. First, if the conditional edge is dominated by an existing requirement constraint, then the algorithm returns true. Second, the algorithm

If This Changes:	Must Back-Propagated Through (Threats)	Updates	Rule:
[-] Req. edge BA	1. any [+] Req. edge CB	[+/-] Req. edge CA	STN(ii)
	2. any Ctg Link CB	[+/-] Req. edge CA	PR
[+] Req. edge AB	1. any [-] Req. edge BC	[+/-] Req. edge AC	STN(i)
	2. any Ctg. Link CB *	[+/-] Cond. edge AC**	PR/UR
	3. any [-] Cond. edge BC of <-t,D>, where D ≠ A	[+/-] Cond. edge AC**	REG(i)
[-] Cond. edge BA of <-t,D>	1. any [+] Req. edge CB, where C ≠ D	[+/-] Cond. edge CA**	REG(i)
	2. any Ctg. Link CB , where B ≠ D	[+/-] Cond. edge CA**	REG(ii)

Table 1 STNU-DBP Rules

* same for both precede or unordered cases , ** convert any conditional edges into requirement edges as required by the UUR.
STN: STN-DBP, UR: Unordered Reduction, UUR: Unconditional Unordered Reduction, Ctg: contingent, Req.: requirement
PR: Precede Reduction, REG: regression

```
function BACK-PROPAGATE(G,u,v)
1    if IS-POS-LOOP( u, v )  return TRUE
2    if IS-NEG-LOOP( u, v ) return FALSE
3    for each threat (x,y) to edge (u,v)
4        apply DBP rules to derive a new candidate edge (p,q)
5        if  (p,q) is conditional
6            if dominated by a Req. edge (p,q) return TRUE
7            convert (p,q) to Req. edge as required by  UUR
8        end if
9        resolve the edge (p,q) with  G
10       if  G is modified
11           if  G is squeezed return FALSE
11           if ¬BACK-PROPAGATE(G,p,q)  return FALSE
12       end
13   end for
14   return TRUE
```

Figure 3 Pseudo-Code for Back-Propagate

converts the conditional edge into a requirement edge as required by the unconditional unordered reduction.

Next the algorithm resolves the candidate edge (p,q) with G by tightening or adding the corresponding edge as necessary. If this resolution modifies a constraint in G (i.e. is not dominated by an existing edge (p,q)), the algorithm checks if this tightening squeezes an uncontrollable duration, then recursively calls BACK-PROPAGATE to resolve the change. After recursively resolving all threats, the algorithm returns true.

The function BACK-PROPAGATE is our Incremental DC maintenance algorithm when a single constraint changes. In order to handle multiple constraint changes we need to apply the BACK-PROPAGATE function to all edges that change.

In the next section we present the Incremental DC Reformulation algorithm that is capable of reformulating unprocessed plans.

Fast Incremental Dynamic Controllability Algorithm

In this section we describe our Fast Incremental-DC Reformulation algorithm (Fast-IDC) which builds upon the BACK-PROPAGATE algorithm presented in the previous section. The Fast-IDC algorithm efficiently reformulates unprocessed plans. We will use the example presented in Figure 4A to describe this algorithm.

The pseudo-code for the Fast-IDC algorithm is shown in Figure 5. If the STNU is dynamically controllable, then the Fast-IDC returns a minimal dispatchable CDGU, otherwise it returns NIL.

First the Fast-IDC algorithm converts the STNU into a CDGU, then computes the pseudo minimal dispatchable graph (PMDG) using the "slow" STN Reformulation Algorithm introduced by [Muscettola 1998]. If an

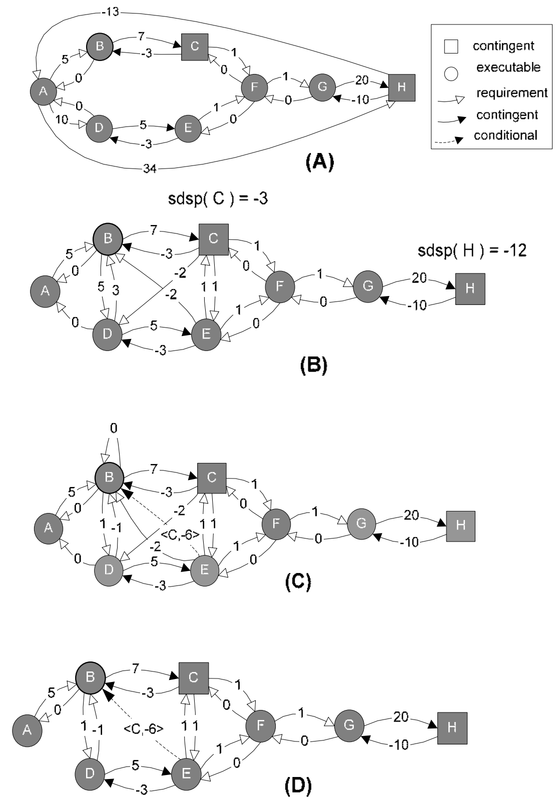

Figure 4 Fast-IDC Example.

inconsistency was detected, the algorithm returns NIL. The minimal pseudo-dispatchable graph for our example is shown in Figure 7-B. This PMDG is both pseudo-dispatchable and contains the fewest number of edges.

The CDGU is only dynamically controllable if it is pseudo-controllable [Morris 2001]. Lines 3 checks if the contingent edges were squeezed during the process of converting the CDGU into a minimal pseudo-dispatchable graph. In our example, all contingent edges remain unchanged; therefore, the CDGU is pseudo-controllable.

Recall that our goal is to reformulate the graph to ensure that the plan can be dynamically executed. This reformulation is done by multiple calls the function BACK-PROPAGATE. The BACK-PROPAGATE function needs to be applied to any edge that may squeeze an uncontrollable duration. Each initial call of BACK-PROPAGATE causes a series of other edge updates. However, they will only update edges closer to the start of the plan. In order to reduce the amount of redundant work, we initiate the back-propagations near the end of the plan first. In order to organize the back-propagations, we need to create a list of contingent timepoints ordered from timepoints that are executed near the end of the plan, to timepoints that are executed near the beginning of the plan. The contingent timepoints are ordered based on their Single-Destination Shortest-Path (SDSP) distance, sdsp(x). Specifically, the contingent timepoints are ordered from

```
function FAST-DC( Γ )
1  G ← STNU_TO_CDGU(Γ)
2  if ¬COMPUTE_PMDG(G)  return NIL
3  if ¬ IS_PSEUDO_CONTROLLABLE (G)  return NIL
4  Compute Bellman_Ford_SDSP( start(G), G )
5  Q ← ordered list of Ctg. T.P. according to the SDSP distances
6  while( ¬Q.IS-EMPTY()  )
7     n ← Q.POP_FRONT()
8     if ¬ BACK_PROPAGATE_INIT( G, n ) , return NIL
9  end while
10 (optional)  COMPUTE_MPDG(G)
11 return G
```

Figure 5 Pseudo-code for FAST-DC

```
function BACK-PROPAGATE-INIT(G,v)
1   for all  pos. edges (u,v) into the Ctg. timepoint v
2     if ¬BACK_PROPAGATE(G,u,v)  return FALSE
3   end for
4   for all outgoing negative edges (v,u) from the ctg timepoint v
5     if ¬BACK_PROPAGATE(G,v,u) return  FALSE
6   end for
7   return TRUE
```

Figure 6 Pseudo-code for BACK-PROPAGATE-INIT

lowest to highest SDSP distances. The SDSP distances are computed in Line 4, and the contingent timepoints are ordered in Q in Line 5. In our example, the two contingent timepoints C and H have SDSP distances of -3 and -12 respectively. Therefore, H comes before timepoint C in the ordered list.

Next the Fast-DC algorithm initiates a series of back-propagations by calling the function BACK-PROPAGATE-INIT. This function initiates the back-propagation by applying all back-propagation rules to ensure that the uncontrollable duration associated with the contingent timepoint *v* is never squeezed during execution. Recall the contingent duration can only be squeezed by incoming positive edges or outgoing negative edges to the contingent timepoint. Lines 1-3 of this initiation function call BACK- PROPAGATE for all incoming positive edges into the contingent timepoint *v* and Lines 4-7 calls BACK-PROPAGATE for all outgoing negative edges from *v*.

Consider the series of back-propagations the Fast-DC algorithm uses to reformulate the CDGU between 7-B and 7-C. The CDGU does not contain threats that may violate contingent timepoint H, so no back-propagations are required. The contingent timepoint C, is threatened by the incoming positive edge EC. The edge EC is back-propagated through BC, resulting in a new conditional edge EB of <C,-6>. This contingent edge is then back-propagated through DE which modifies the requirement edge DB to -1. This negative requirement edge is then back-propagated through edge BD resulting in the edge BB of distance 4. This thread of back-propagation terminates here because of a positive self-loop.

The contingent timepoint C is also threatened by the outgoing negative edge CD of length -2. This negative requirement edge CD is back-propagated through BC, which sets BD = 1. This positive requirement edge is then back-propagated through the negative edge DB, resulting in modifying the self-looping edge BB to 0. This thread of back-propagation is then terminated. The resulting dispatchable CDGU is shown Figure 4C. The back-propagation did not introduce an inconsistency; therefore, the original STNU is dynamically controllable.

The (optional) last step of the Fast-DC algorithm trims the dominated (redundant) edges from the CDGU. This is done by calling the basic STN reformulation algorithm. The resulting graph is a minimal dispatchable CDGU which can be executed by the dispatching algorithm introduced by [Morris 2001]. For example, the minimal dispatchable CDGU for the sample group plan is shown Figure 4D.

In this section, we presented an efficient algorithm to reformulate an STNU into a dispatchable CDGU.

Run Time Complexity of the FAST-IDC Algorithm

In this section describes some experimental results for our Fast-IDC algorithm. The FAST-IDC algorithm was implemented in C++ and run on a set of randomly generated STNUs that contained between 10 to 50 activities (20-140 timepoints) interconnected by a set of random (yet locally consistent) requirement edges. In our trials, 50% of the activities were uncontrollable.

Figure 8 shows the experimental run time of the Fast-DC and DC algorithm introduced by [Morris 2001] for successful reformulations, plotted against the number of activities in the STNU. The tests were run on a 1 GHz Pentium IV processor with 512 MB of RAM. The data label DBP represents the time the algorithm spent in the BACK-PROPAGATE function.

The most interesting result is the speed at which the algorithm performed the back-propagations. Recall when maintaining the dispatchability of our plan we only need to call the BACK-PROPAGATE function. Thus maintaining the dispatchability is very efficient as shown the by the data labeled DPB in Figure 8.

Our test also shows that our Fast-DC algorithm experimentally runs in $O(N^3)$ when reformulation unprocessed plan. This is not surprising if you consider the overall structure of the Fast-IDC algorithm as follows.

1. Compute PMDG	$\Theta(N^3)$
2. Check for Pseudo-Controllability	O(E)
3. Run SSSP	O(NE)
4. Back-Propagation	polynomial
5. (optional) Re-compute PMDG	$\Theta(N^3)$

Our Fast-DC algorithm is dominated by the “slow” STN reformulation algorithm in step 1. Our FAST-DC algorithm can be directly improved by using the “fast”

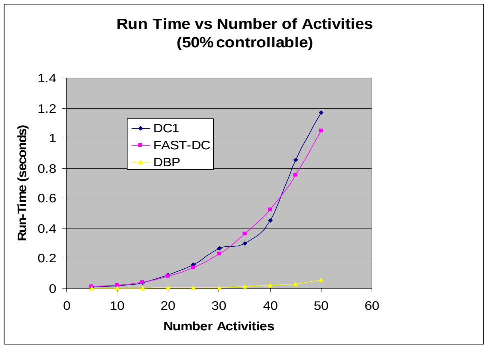

Figure 8 Run Time Complexity

STN reformulation algorithm introduced by [Tsarmardinos 1998], which runs in O(NE + N^2 log N) time.

Future Work

Currently our incremental dispatchability algorithm is only capable of maintaining the dispatchability when the values of the constraints are tightened. We are currently investigating an improved algorithm to handle the case when new activities are added or removed from the plan or when the contingent activities timebounds are relaxed rather than tightened.

References

[CLR 1990] T.H. Cormen, C.E. Leiserson and R.L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 1990.

[Dechter 1991] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61-95, May 1991.

[Morris 2001] P. Morris, N. Muscettola, and T, Vidal. Dynamic Control of plans with temporal uncertainty. In: *Proc.* IJCAI-01.

[Muscettola 1998] N. Muscettola, P. Morris, and I. Tsamardinos. Reformulating temporal plans for efficient execution. In *Proc. Of Sixth Int. Conf. on Principles of Knowledge Representation and Reasoning* (KR '98), 1998.

[Tsamardinos 1998] I. Tsarmardinos, N. Muscettola, and P.Morris. Fast transformation of temporal plans for efficient execution. AAAI-98, 1998.

[Vidal 1996] T. Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraint networks dedicated to planning. In *Proc.* ECAI-1996.

[Vidal 1999] T. Vidal and H. Fargier. Handling contingency in temporal constraint networks: from consistencies to controllabilities. Journal of Experimental & Theoretical Artificial Intelligence, 11:23-45, 1999