A Fast Incremental Algorithm for Maintaining Dispatchability of Partially Controllable Plans

Julie Shah, John Stedl, Brian Williams, and Paul Robertson
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
32 Vassar St. Room 32-G275, Cambridge, MA 02139
julie_a_shah@csail.mit.edu, johnstedl@gmail.com, williams@mit.edu, paulr@csail.mit.edu

Abstract

Autonomous systems operating in real-world environments must be able to plan, schedule, and execute missions while robustly adapting to uncertainty and disturbances. Previous work on dispatchable execution increases the efficiency of plan execution under uncertainty by introducing a temporal plan dispatcher that schedules events dynamically in response to disturbances, and a compiler that reduces a plan to a *dispatchable* form that enables real-time scheduling. However, this work does not address the situation where response requires modifying the plan in real-time. In these situations, after the autonomous system replans, compilation to dispatchable form must occur in near real-time.

The key contribution of this paper is a fast Incremental Dynamic Control algorithm (IDC) for maintaining the dispatchability of a partially controllable plan, in response to incremental plan modifications by an online planner. IDC is developed as a set of incremental update rules that exploit the structure of the plan in order to efficiently propagate the effects of constraint loosening and tightening throughout the plan. IDC exhibits an order of magnitude improvement in compile time over the state of the art non-incremental algorithm applied to randomly generated problems. Its practicality is demonstrated on plans for coordinating rovers within the authors' hardware test-bed.

Introduction

Often, autonomous agents that operate in real-world environments must be able to plan, schedule, and execute missions while robustly anticipating and adapting to uncertainty and disturbances. Typically an agent only controls the timing of a subset of a plan's events; timing of the other events is controlled exogenously by nature or other agents. For example, a Mars rover can control when it starts driving to a rock; however, its precise arrival time is influenced by environmental factors. To achieve successful execution of a partially controllable plan, the scheduler must guarantee that all temporal constraints are satisfied, even though some events are uncontrollable. Since it is difficult to provide such a guarantee without any knowledge about the behavior of uncontrollable events, the scheduler exploits a model, called a *simple temporal network with uncertainty (STNU)* [Vidal 1996, Vidal and Fargier 1999], to explicitly represent plan uncertainty by bounding the behavior of uncontrollable events.

The domain of application for STNUs is embedded systems, such as airplanes and robotic systems, which perform scheduling within their controller. The field of STNU dispatching focuses on embedded control applications in which the scheduler must satisfy hard scheduling constraints while accommodating disturbances. Applications include scheduling within the avionics processor of commercial aircraft [Tsamardinos et al. 1998], and control of space probes [Muscettola et al. 1998b], autonomous air vehicles [Stedl 2004], and walking robots [Hofmann et al. 2006].

For a given STNU, it may not be possible to generate a static schedule a priori that guarantees successful plan execution over all possible execution times of uncontrollable events. If a static schedule does exist, it may be overly conservative in plan completion time. This problem is addressed through *dynamic control* [Vidal 2000], a strategy that schedules controllable events online just before they are executed. This strategy exploits the fact that uncertainty associated with past uncontrollable events is eliminated, allowing the scheduler to be less conservative in the schedules it generates. Given a set of temporal constraints over controllable and uncontrollable events, and observations of past events, a *dynamic control* strategy generates a schedule online that guarantees the temporal constraints of the plan are satisfied.

Dynamic control is achieved through *dispatchable execution* [Muscettola 1998, Morris et al. 2000], the incremental computation of feasible schedules performed through constraint propagation, to update the network as new information is received. The timing of executed events is propagated throughout the network to ensure that time windows of later events are appropriately narrowed to satisfy timing constraints of the plan. Dynamic control of STNUs, achieved through dispatchable execution, is domain independent, and applicable to online scheduling for systems that have uncontrollable events and must satisfy hard scheduling constraints.

To achieve the goal of scheduling STNUs in real-time, [Morris et al. 2001] introduced a *dynamic controllability (DC)* algorithm to 1) determine if a dynamic control strategy exists for an STNU, and if so, 2) compile the STNU to a *dispatchable* form which reduces the amount of propagation necessary during execution, making it possible to schedule in real-time. The dispatchable plan is precompiled before plan execution, and is then used by a *dispatcher* to schedule quickly online.

This paper focuses on the additional technical challenge of responding, in real-time, to disturbances that require modifying the plan. To sustain operation during a critical mission phase, the agent may need a way to quickly replan and then recompile the modified plan into a dispatchable form. Significant progress has been made on the first problem – fast replanning. Efficient solutions include the use of local repair [Zweben 1993, Rabideau et al. 1999], and incremental search [Shu 2003, Effinger 2006]. However, existing compilation algorithms are insufficient for real-time performance.

We confront the challenge of real-time compilation based on the observation that during replanning, typically only a small portion of a plan is modified. Our compilation algorithm improves efficiency substantially by incrementally updating the dispatchable plan in response to plan changes in the spirit of other incremental algorithms for truth maintenance [Doyle 1979] and informed search [Koenig et al. 2001]. Current DC compilation algorithms repeatedly compute an all-pairs shortest path (APSP) graph and then check all possible triangles in the network for reductions. In contrast, our IDC algorithm maintains dispatchability as constraints in the plan are tightened (or added) and loosened (or removed). This is achieved through a set of incremental update rules that exploit the causal structure of the plan to efficiently propagate the effect of each changed constraint throughout the network.

This paper presents our incremental compilation algorithm and its empirical validation. First, we describe a practical scenario involving the coordination of rovers as an example for the rest of the paper. Next we review STNUs and the DC algorithm. We then develop our incremental algorithm in two parts: we present how to maintain dispatchability for the case when a constraint is tightened or added to the plan, and then for the case where constraints are loosened or removed from the plan. Finally, we present empirical results comparing the incremental algorithm to the DC algorithm and conclude.

Practical Scenario

Consider a two rover scenario used to demonstrate online replanning, DC compilation and dispatching on a hardware test-bed [Robertson and Williams 2005] (**Fig.1**). In this scenario, the two rovers cooperatively search for science targets in a simulated Martian environment. Rover 1 drives to location 3, via a route through location 1. At location 3, Rover 1 surveys the area for interesting science targets. Simultaneously, Rover 2 drives to location 4, via a route through location 2. At location 4, Rover 2 surveys the area for interesting science targets. When both rovers finish surveying their respective areas, they rendezvous at the same time at location 0. The plan corresponding to this description involves uncontrollable events; for example, the time the rovers spend "finding targets" is uncertain since the rovers may find a target right away or use the maximum allotted time without finding a target.

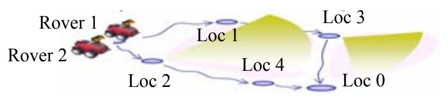


Figure 1: Cooperative Rover Scenario

Background

A Simple Temporal Network with Uncertainty (STNU) [Vidal and Fargier 1999] is an extension of an STN [Dechter 1991] that distinguishes between controllable and uncontrollable events. An 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: *requirement links* and *contingent links*. A requirement link specifies a constraint on the duration between two timepoints. A contingent link models an uncontrollable process whose uncertain duration, ω, may last any duration between the specified lower and upper bounds. All contingent links terminate on a *contingent timepoint* whose timing is controlled exogenously. All other timepoints are called *requirement timepoints* and are controlled by the agent. **Fig.2** presents an STNU representing the cooperative rover scenario of the preceding section.

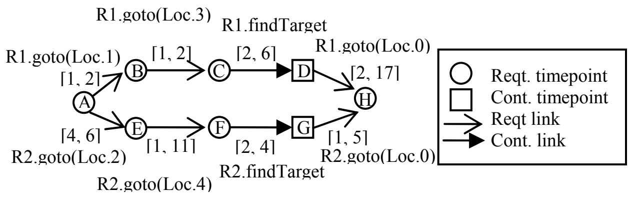


Figure 2: STNU for Cooperative Rover Scenario

This STNU must be checked to determine if a dynamically controllable execution strategy exists, and then compiled into a *dispatchable* form. Once the plan is *dispatchable*, the dispatcher consistently and efficiently schedules timepoints through local propagation of timebounds [Muscettola 1998, Morris et al. 2000]. We now review how STNUs are compiled into dispatchable form and present the dispatchable form for the rover scenario (**Fig.5**).

To support efficient inference, an STNU is mapped to an equivalent distance graph [Dechter 1991], which we call a Distance Graph with Uncertainty (DGU). Each link of the STNU, containing both lower and upper bounds, is converted to a pair of edges in the DGU. One edge in the forward direction is labeled with the value of the upper time bound, and one edge in the reverse direction is labeled with the negative of the lower time bound. The distinction between contingent and requirement edges is maintained. **Fig.3** presents the cooperative rover scenario DGU. Edge BC [5, 7] in **Fig.2** is converted to a pair of edges in **Fig.3**, where the forward edge BC value is 7, and the reverse edge CB value is -5.

An STNU is *consistent* only if its associated distance graph 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 is not sufficient to guarantee dynamic controllability, meaning that there is enough flexibility in the plan to compensate at execution time for temporal uncertainty in the plan.

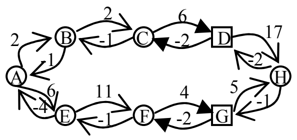


Figure 3: DGU for Cooperative Rover Scenario

The dynamic controllability (DC) algorithm introduced by [Morris et al. 2001] reformulates the DGU to ensure that each uncontrollable duration, ω_i, is free to finish any time between $[l_i,u_i]$, as specified by the contingent link, C_i. We review the three steps of the DC algorithm. The first step (1) computes the APSP-graph of the DGU using the Floyd-Warshall algorithm [CLR 1990] in order to expose implicit temporal constraints. Exposing implicit constraints is necessary to ensure events are scheduled in the proper order, and with requisite temporal distances between events. If the exposed constraints imply strictly tighter bounds on an uncontrollable duration, then that uncontrollable duration is *squeezed* [Morris et al. 2001] and the plan is not dynamically controllable. In this case there exists a *situation* [Vidal 1999] where the outcome of the uncontrollable duration results in no feasible schedule of controllable events to satisfy the STNU. An STNU is *pseudo-controllable* [Morris et al. 2001] if it is both temporally consistent and none of its uncontrollable durations are squeezed.

However, even if an STNU is pseudo-controllable, the uncontrollable durations may be squeezed at execution time [Morris et al. 2001] as follows. When the dispatcher executes a timepoint, it fixes the value of the timepoint. Updating the implicit constraints based on this value may then squeeze, meaning imply tighter bounds, on a contingent link. To avoid squeezing uncontrollable durations, the DC algorithm, Step (2) adds constraints to the plan. The constraints take the form of simple temporal constraints and conditional constraints (or "wait" constraints) and are applied according to the *precede, unordered, and unconditional unordered reduction rules* described in [Morris et al. 2001]. We review the reduction rules since they are important to understanding our incremental update rules.

Consider the triangular DGU shown in **Fig.4**. Assume the DGU is both pseudo-controllable and in APSP-form.

When C is executed before B ($v \le 0$, $u<0$), the dispatcher will never know the execution of the contingent timepoint B when it needs to schedule timepoint C. To maintain dynamic controllability, the dispatcher must avoid a situation in which uncontrollable duration AB is squeezed due to propagations from CB and BC during dispatching. To ensure this does not happen, the dispatcher must constrain the temporal relationship between timepoints A and C such that, no matter how long uncontrollable duration AB takes within [x, y], timepoint C can be executed to satisfy constraints CB and BC. The precede reduction achieves this by tightening constraints AC and CA as follows.

Definition (Precede Reduction [Morris et al. 2001]**)** *If $v \le 0$, $u<0$, tighten AC to x-u, and edge CA to v-y.*

When the execution of B and C are unordered ($v \ge 0$ and $u \le 0$), the unordered reduction uses a *conditional constraint* to prevent propagations from possibly squeezing the uncontrollable duration AB during dispatching. If C is executed before B (as in the precede reduction), constraint CA must be tightened to ensure that no matter how long uncontrollable duration AB takes within [x, y], constraint CB will be satisfied. If B is executed before C, then the dispatcher knows the execution time of B when scheduling timepoint C, and tightening CA is not necessary.

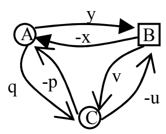


Figure 4: Triangular Distance Graph with Uncertainty (DGU)

Definition (Unordered Reduction [Morris et al. 2001]**)** *If $v \ge 0$ and $u \le 0$, apply a conditional constraint CA of <B, v-y>.*
For example, in **Fig.5** conditional edge GC labeled <-3, D> specifies that G must wait at least 3 time units after C executes or until D executes, whichever comes first. We call a DGU containing a set of conditional constraints a *Conditional Distance Graph with Uncertainty* (CDGU).

If the conditional edge created by the unordered reduction requires that C is always executed before B, then the edge is unconditional. The unconditional unordered reduction describes when to convert the conditional edge into a requirement edge.

Definition (Unconditional Unordered Reduction [Morris et al. 2001]**)** *Given an 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 into a requirement edge CA with distance –x.*

Step (3) of the DC algorithm applies the rules for *regression* to the conditional constraints in the CDGU. The rules for *regression*, described in [Morris et al. 2001], add constraints to the CDGU to ensure that the conditional constraints created by the reduction rules are not violated at execution and are satisfied for all outcomes of uncontrollable events. We review the regression rules since they are also important to understanding our incremental update rules.

Lemma (Regression [Morris et al. 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 $w \ge 0$ and $D \ne B$, 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 $B \ne C$, then we can deduce a conditional constraint DA of <x+t, B>.
The rules for regression are applied recursively to all conditional constraints in the CDGU, until no more regressions are possible.

If the CDGU is modified during Steps (2) or (3), then the DC algorithm loops to Step (1) and computes the APSP-graph again to propagate the effect of the added constraints throughout the network. The DC algorithm iterates through Steps 1-3 until the CDGU is found to be inconsistent, not pseudo-controllable, or else the CDGU is not modified. If the CDGU is not modified, it is then trimmed of all *dominated edges* to reduce propagations at execution time. An edge is dominated if in all possible executions, another edge exists that always propagates a tighter bound. [Muscettola 1998] showed that the dominated edges can be removed without adversely affecting the ability of the dispatcher to dynamically execute the network. The trimmed CDGU for the cooperative rover scenario is presented in **Fig.5.**

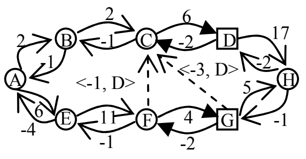


Figure 5: Dispatchable CDGU for Cooperative Rover Scenario

Incremental Algorithm for Maintaining Dispatchability

In this section, we present our incremental algorithm, IDC, which enables the agent to quickly maintain dispatchability after a fast replanner modifies a subset of the constraints. IDC uses incremental update rules in the spirit of incremental search algorithms [Koenig and Likhachev 2001], and employs a *set of support* similar to truth maintenance systems [Doyle 1979]. The key innovation of our algorithm is a unified set of incremental update rules that exploit the causal structure of the plan to interleave and efficiently apply the different types of propagation in the DC algorithm. This is in contrast to how the DC algorithm repeatedly computes the all-pairs shortest path (APSP) graph and repeatedly checks all possible triangles in the network for reductions.

Our IDC algorithm maintains dispatchability when constraints in the plan are both tightened (or added) and loosened (or removed). We first address the problem of maintaining dispatchability when constraints are tightened. We then provide an intuitive explanation for the difference between maintaining dispatchability when constraints are tightened versus loosened, and address the problem of maintaining dispatchability when constraints are loosened.

Constraint Tightening

The speed of our IDC algorithm is derived from exploiting the causal structure of a dispatchable plan to propagate constraint modifications throughout the plan. We introduce a technique we call *dispatchability back-propagation (DBP)* to resolve STN constraint tightening. We then present a unified set of incremental update rules derived from DBP, *reduction*, and *regression* rules to resolve the constraint tightening in an STNU; by *resolve* we mean to expose implicit constraints implied by the constraint tightening, to check consistency, and to remove any case in which an uncontrollable event may be squeezed.

We develop *dispatchability back-propagation (DBP)* by exploiting the dispatchability of plans [Muscettola 1998]. 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 past scheduling decisions and observations. This is possible since the compilation process has already imposed the constraints of future events on the current event. Recall that executing an event is equivalent to fixing the value of a timepoint. During execution, the dispatcher ensures that future scheduling decisions are consistent with these fixed values by propagating information at execution time. To incorporate scheduling decisions into the dispatchable graph, when a timepoint A is executed, upper-bound updates are propagated to A's immediate neighbors 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.

To maintain the dispatchability of the STN when a constraint is tightened by a fast replanner, we only need to make the modified constraint consistent with past scheduling decisions, since during execution, the bounds on events are only influenced by preceding events. This involves eliminating all assignments to the current event that are inconsistent with the STN and past scheduling decisions. When an edge X is tightened, it only needs to be made consistent with the set of edges that may cause an inconsistency with the time window update propagated by edge X at execution time. These edges are called *threats.* The same reasoning applies if a constraint is added to the network, since adding a constraint can be thought of as a constraint tightening that goes from a bound of positive or negative infinity to a finite value. We use *DBP* to refer to the process of ensuring an STN edge tightening or addition is consistent with the past scheduling decisions,

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 positive edge AB propagates an upper bound to B of ubB = T(A) + d(AB). A non-positive edge BC propagates a lower bound to B of lbB = T(C) - d(BC). At execution time, changing AB will be consistent if ubB >= lbB for any C, or T(A) + d(AB) >=

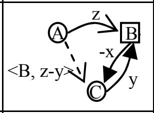
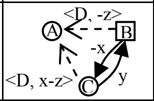
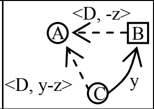
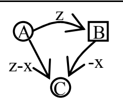
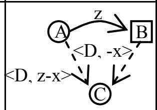
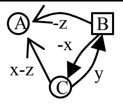
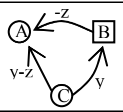
T(C) - d(BC), which implies T(C) - T(A) < d(AB) + d(BC). Adding an edge AC of d(AB) + d(BC) to G encodes this constraint. Similar reasoning applies for case (ii) when a negative edge changes.

Recursively applying rules (i) and (ii), when an edge is tightened in a dispatchable distance graph, will either expose a direct inconsistency or result in a dispatchable graph[1]. The key feature of DBP is that it only requires a subset of the edges be checked to ensure the modified constraint is consistent, rather than all edges when the APSP-graph is computed.

For the DC algorithm, in addition to computing implied constraints by generating the APSP-graph, the algorithm applies *reduction* and *regression* rules to ensure that uncontrollable durations are not squeezed at execution time. Likewise, to resolve squeezing in our IDC algorithm, we interleave the DBP rules with incremental updates rules derived from the *reduction* and *regression* rule sets. This unified set of incremental update rules (described in **Table 1)** is used to ensure dynamic control. Each incremental update rule differs, depending on the types of edges involved, the signs of the edge distances, and the relative direction of the edges. A DGU consists of five types of edges: positive and negative requirement edges, positive and negative contingent edges, and negative conditional edges. The incremental update rules describe the propagation of three of these edge types: negative requirement edges, positive requirement edges, and negative conditional edges - these are the only three types of edges that may be added or modified during compilation. (Any positive conditional edge is converted to a requirement edge by the unconditional unordered reduction rule.)

Fig.6 shows the pseudo-code for BACKPROPAGATE-TIGHTEN, which uses the unified set of incremental update rules to maintain dispatchability of a conditional distance graph with uncertainty (G) when a subset ($e_1 \ldots e_n$) of edges are tightened or added to the graph. Since the incremental update rules propagate edge updates towards the start of the plan, we reduce the amount of redundant work in BACKPROPAGATE-TIGHTEN by initiating propagations near the end of the plan first. In Line 1, the relevant timepoint for each new or modified edge is ordered according to single-destination shortest-path (SDSP), from lowest to highest. IDC chooses the relevant timepoint based on how the edge is back-propagated, it: (1) uses the source timepoint of the edge if the edge is conditional or the edge value is less than or equal to zero, and (2) uses the target timepoint if the edge value is greater than zero. Then, for each edge in the ordered list, IDC checks if edge e_i is a loop (i.e. starts and ends at the same timepoint). A positive loop indicates no additional propagations are required; IDC skips e_i and proceeds with the next modified edge in the ordered list. A negative loop indicates an inconsistency; IDC returns false. Next, the algorithm resolves all possible *threats* to e_i by applying the necessary incremental update rules (**Table 1**).

Table 1: Incremental Update Rules

#	Graphical Description	Pre-conditions	Post- conditions	Deri-vation
1		Propagate incoming (+) reqt. edge AB changes through any outgoing (-) cont. edge BC.	Create cond. edge AC. *	Un-ordered Reduction **[Morris 2001]**
2		Propagate outgoing cond. edge BA through any incoming (+) cont. edge CB.	Create cond. edge CA. *	Regress-ion **[Morris 2001]**
3		Propagate outgoing cond. edge BA through any in-coming (+) reqt. edge CB except when D=C.	Create cond. edge CA. *	Regress-ion **[Morris 2001]**
4		Propagate incoming (+) reqt. edge AB through any outgoing (-) reqt. edge BC.	Create new reqt. edge AC if none exists or tighten if (z-x)< existing edge.	DBP(i)
5		Propagate incoming (+) reqt. edge AB through any outgoing cont. edge BC except A=D.	Create cond. edge AC. *	Regress-ion **[Morris 2001]**
6		Propagate out-going (-) reqt. edge BA through any incoming (+) cont. edge CB.	Create new reqt. edge CA if none exists or tighten if (x-z)< existing edge.	Precede Reduction **[Morris 2001]**
7		Propagate out-going (-) reqt. edge BA through any incoming (+) reqt. edge CB.	Create new reqt. edge CA if none exists or tighten if (y-z)< existing edge.	DBP(ii)

*If necessary, apply the Unconditional Unordered Reduction [Morris 2001]. If resulting req. edge is new or tighter the existing edge, update reqt. edge.

```
function BACKPROPAGATE-TIGHTEN(G, e1…en)
1.  Q ← order e1…en by SDSP, lowest to highest
2.  for each modified edge ei in ordered Q
3.      if IS-POS-LOOP(ei) then SKIP ei
4.      if IS-NEG-LOOP(ei) return FALSE
5.      for each incremental update rule propagating ei
6.          if edge zi in G is modified or created
7.              if G is squeezed return FALSE
8.              if ¬BACKPROPAGATE-TIGHEN(G, zi) return FALSE
9.          end
10.     end for
11. end for
12. return TRUE
```

Figure 6: Pseudo-code for BACKPROPAGATE-TIGHTEN

BACKPROPAGATE-TIGHTEN recursively applies the incremental update rules until either a direct inconsistency is detected or until no propagation remains. Consider a tightening of HD in **Fig.5** to -10. Rule 7 is applied to propagate HD through GH, resulting in a new edge GD of -5. Rule 6 is used to propagate the new edge GD through FG, resulting in a new edge FD of -3. Other propagation

[1] Note that application of the STN-DBP rules does not result in an APSP-graph, and instead relies on the properties of a disptachable graph to make consistent assignments to timepoints at execution. Therefore, STN-DBP is not the basis for a fully-dynamic all-pairs-shortest-path algorithm [Demestrescu 2004], and is not analogous to an incremental local consistency algorithm in the constraint satisfaction framework [Bessiere, C. 1994].

related to these modified and new edges do not result in edge tightenings, and no other propagation remain.

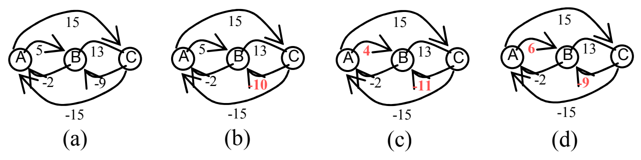


Figure 7: (a) distance graph of an STN, (b) associated APSP graph, (c) tightening edge AB to 4, (d) loosening of edge AB to 6.

Tightening vs. Loosening Constraints

In this section, we use a simple STN example to provide an intuitive explanation for the difference between maintaining dispatchability when constraints are tightened versus loosened. Consider the distance graph of a STN shown in **Fig.7a**. The associated APSP graph is shown in **Fig.7b.** The APSP computation is used to reduce a STN into dispatchable form [Muscettola 1998].

When a constraint is tightened, this change needs only to be made consistent with the past scheduling decisions and the dispatcher will then ensure that this constraint change is consistent with the future at execution time. To illustrate this, consider what happens when edge AB is tightened from 5 to 4 (**Fig.7c**). As long as this change is consistent with the past (it is), then the dispatcher is able to compensate for the tightening of AB by choosing the appropriate execution time of C within the range [11, 13] after B.

In contrast, consider what happens when edge AB is loosened from 5 to 6 (shown in **Fig.7d**). Timepoint C must now be executed with a new lower bound of 9 time units after B to ensure that C occurs exactly 15 time units after A. The value 9 is not within the range [10, 13]; a situation may arise where the dispatcher cannot compensate for the loosening of AB using the dispatchable form. However, remember that the BC timebound before the APSP computation was [9, 13]. The value of edge CB was tightened from -9 to -10 during the APSP computation using edge values CA and AB as support. Since the value of AB has changed, CB can revert back to -9. Dispatchability is maintained as long as the AB value of 6 and CB value of -9 are consistent with previous timepoints.

This simple STN example shows that it is necessary to maintain a list of edge value support to identify the influence of loosening a temporal constraint. A similar argument can be made for maintaining a list of dominated edge support. Support lists are also used if a constraint is removed from a network, since this is an edge loosening from a finite value to positive or negative infinity.

Support lists, also called *set of support*, were first used for incremental updates in truth-maintenance systems [Doyle 1979]**,** in order to record justification, recognize inconsistencies, and remember derivations. In this spirit, IDC, like other incremental graph algorithms, uses support when constraints are loosened to identify edge values that are no longer valid and revert them to supported values. IDC is unique in that it also uses support to identify trimmed edges that are no longer dominated and to add them back into the CDGU.

Constraint Loosening

In this section, we first discuss how to build a set of support during plan compilation, before dispatching. We then discuss how the set of support is used to efficiently maintain dispatchability when constraints in the plan are loosened (or removed) during execution.

We assume that the DC algorithm is used to precompile the plan before dispatching, and our IDC algorithm is then used for maintaining dispatchability in response to incremental plan modifications made by an online planner. Since the IDC algorithm relies on a set of support to identify the influence of loosening a temporal constraint, we first discuss how we modify the DC algorithm to build set of support.

The DC algorithm must maintain an *edge value support list* to determine the set of edge values no longer valid due to a modified constraint, and a *dominated edge support list* to determine the set of edges no longer dominated due to a modified constraint. A new edge value support is added to the set of support when: (1) an edge value is updated in the Floyd-Warshall algorithm, (2) a conditional or requirement edge is created or updated using the *reduction* rules, or (3) during *regression*. The following information is recorded: (a) the edge that is updated, (b) the new value of the updated edge, (c) the conditional node (if applicable) (d) the supporting edges used to calculate the new edge value, and (e) information describing how each supporting edge was used to derive the new edge value. The derivation of a new or modified edge depends on either the *value, positivity,* or *negativity* of a supporting edge. For example, in **Fig.7b,** CB is supported by the *values* of CA and AB.

A *dominated edge support list* is built when the CDGU is trimmed of all dominated edges. A dominated edge support is added to the set of support for each edge trimmed from the CDGU, and includes the following information: (1) the edge that was trimmed and (2) the supporting edges used to determine domination.

Using this set of support, we adapt IDC's BACKPROPAGATE-TIGHTEN algorithm to maintain dispatchability when constraints are loosened. The pseudo-code for IDC's BACKPROPAGATE-LOOSEN is shown in **Fig.8-10**. This algorithm maintains dispatchability of a conditional distance graph with uncertainty (G) when a subset ($e_1 \ldots e_n$) of edges are loosened or removed, using the dominated edge support list (L_D) and the edge value support list (L_E) built during the initial compilation. As in BACKPROPAGATE-TIGHTEN, modified edges are ordered according to SDSP to reduce the amount of redundant work. Each edge e_i in the ordered list is checked whether it is a loop (i.e. starts and ends at the same timepoint). A positive loop indicates no propagations are required; IDC skips e_i and proceeds with the next modified edge in the ordered list. A negative loop indicates an inconsistency; IDC returns false. Next, the LOOSENCONSTRAINTS and BACKPROPAGATE methods are called to identify undominated edges and unsupported edge values, and to propagate the effects of new or modified edges using the incremental update rules.

The method LOOSENCONSTRAINTS uses set of support L_D and L_E to identify the influence of the loosened constraint z_i. First, L_D is used to identify any edges marked as dominated that may now no longer be dominated due to the change in z_i, and orders them according to SDSP, lowest to highest (Line 1). Next, each potentially undominated edge d_i in the ordered list is checked to determine whether there is another valid reason it is still dominated. If the edge is still dominated, then L_D is updated with the new support (Line 4). Otherwise, the undominated edge is added back to the CDGU and removed from L_D (Lines 6, 7). Since adding a constraint to the CDGU is a special case of tightening a constraint, the method BACKPROPAGATE is then called to recursively propagate the resulting modification throughout the network (Line 8). Note that BACKPROPAGATE is very similar to BACKPROPAGATE-TIGHTEN, except that BACKPROPAGATE requires extra work to maintain current set of support.

Next, L_E is used to identify any edges with values that now may be unsupported due to the change in z_i, and orders them according to SDSP, lowest to highest (Line 11). Each unsupported edge v_i in the ordered list is checked to determine whether it represents the current edge value in G, or a previous edge value. If the value is *not* current, then the edge value support is removed from L_E to prevent the edge from later being reverted to an unsupported edge value. If the value *is* current, then the edge v_i is reverted to the last supported value found in L_E (Line 13), and the unsupported edge value is removed from L_E (Line 14). In the case that no last supported edge value is found in L_E, the edge v_i is reverted to the value found in the original CDGU before compilation, and if the edge v_i did not exist in the original CDGU, then the edge is removed from the network. Reverting or removing v_i from the CDGU results in a constraint loosening, and LOOSENCONSTRAINTS must be called to identify the influence of the loosened edge v_i (Line 15). BACKPROPAGATE is then called to recursively propagate the modification of v_i throughout the network (16). Recursive calls to LOOSEN-CONSTRAINTS and BACKPROPAGATE revert the CDGU to a previous stage of compilation to capture the influence of the loosened constraint, and apply the incremental update rules to all edge modifications to maintain dispatchability.

Consider applying BACKPROPAGATE-LOOSEN to the cooperative rover scenario CDGU in **Fig.5**. The plan is dispatched to the rovers and executed up to time 4, when Rover 1 is about to traverse to the rendezvous point (Loc.0). At this point, Rover 1 discovers that the selected path is unexpectedly muddy and estimates that it may take up to 20 time units to reach Loc.0 (i.e. DH is loosened to 20). Calling LOOSENCONSTRAINTS for DH does not yield any undominated edges or unsupported edge values. DH is then back-propagated using the incremental update rules. Rule 4 is applied to propagate DH through HD, yielding edge DD with a value of 18. A positive self loop means no inconsistency was detected and no more back-propagations remain for this threat. Rule 4 is applied again to propagate DH through HG, yielding edge DG with a value of 16. The algorithm continues through a number of further propagations, and in this example results in a dispatchable network. If dispatchability could not be maintained, a fast re-planner would form a contingency plan to circumvent the muddy route, and IDC would then be used to remove and add constraints to efficiently compile the new plan into dispatchable form.

```
function BACKPROPAGATE-LOOSEN (G, e_1…e_n, L_D, L_E)
1.  Q← order e_1…e_n by SDSP, lowest to highest
2.  for each modified edge e_i in ordered Q
3.      if IS-POS-LOOP(e_i) then SKIP e_i
4.      if IS-NEG-LOOP(e_i) return FALSE
5.      if ¬LOOSENCONSTRAINTS (G, e_i, L_D, L_E) return FALSE
6.      if ¬BACKPROPAGATE (G, e_i, L_D, L_E) return FALSE
7.  end for
8.  return TRUE
```

Figure 8: Pseudo-code for BACKPROPAGATE-LOOSEN

```
function LOOSENCONSTRAINTS (G, z_i, L_D, L_E)
1.   D ← IDENTIFY-AND-SDSP-ORDER_UNDOMINATED_EDGES (L_D, z_i)
2.   for each Undominated Edge d_i in ordered list D
3.       if OTHER-REASON-STILL-DOMINATED(d_i, G)
4.           UPDATE-DOMINATED-SUPPORTS(d_i, L_D, G)
5.       else
6.           add undominated edge d_i back to CDGU
7.           REMOVE-FROM-DOMINATED-SUPPORTS(d_i, L_D)
8.           if ¬BACKPROPAGATE (G, d_i, L_D, L_E) return FALSE
9.       end if
10.  E ← IDENTIFY-AND-SDSP-ORDER_UNSUPPORTED_EDGES (L_E, z_i)
11.  for each Unsupported Edge Value v_i in ordered list E
12.      if EDGE-VALUE-CURRENT(v_i, G)
13.          revert unsupported edge v_i to previously supported edge value
14.          REMOVE-FROM-EDGE-VALUE-SUPPORTS(v_i, , L_E)
15.          if ¬LOOSENCONSTRAINTS (G, v_i, L_D, L_E) return FALSE
16.          if ¬BACKPROPAGATE (G, v_i, L_D, L_E) return FALSE
17.      else
18.          REMOVE-FROM-EDGE-VALUE-SUPPORTS(v_i, , L_E)
19.      end if
20.  end for
21.  return TRUE
```

Figure 9: Pseudo-code for LOOSENCONSTRAINTS

```
function BACKPROPAGATE (G, e_i, L_D, L_E)
1.  if IS-POS-LOOP(e_i) return TRUE
2.  if IS-NEG-LOOP(e_i) return FALSE
3.  for each incremental update rule propagating e_i
4.      if edge z_i in G is modified or created
5           add Edge Value Support for z_i to L_E
6.          remove Dominated Edge Support for z_i from L_D if necessary
7.          if G is squeezed return FALSE
8.          if ¬BACKPROPAGATE (G, z_i, L_D, L_E) return FALSE
9.      end if
10. end for
11. return TRUE
```

Figure 10: Pseudo-code for BACKPROPAGATE

Empirical Validation

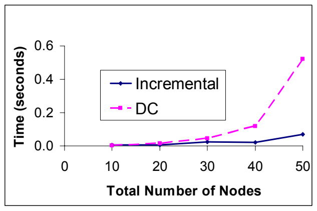
We empirically validated our IDC algorithm by comparing runtime with our implementation of the DC algorithm.

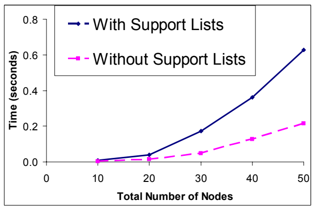
For basis of comparison, we randomly generated STNUs of various sizes of n = 10, 20, 30, 40, and 50 nodes. The STNUs were created by generating n/2 *activities*, each with a start timepoint *S* and end timepoint *E*. An upperbound time constraint between each *S* and *E* was then randomly chosen between [1, max_duration =10], and a lowerbound time constraint was randomly chosen between [0, upperbound] so that the duration would be nonzero and locally consistent. The link and end timepoint of each activity was assigned as contingent with 0.3 probability. To derive constraints among activities, we randomly placed each activity in a 2D plan space similar to a simple scheduling timeline, where overlapping activities represent concurrent activities. Requirement links with generated locally consistent values were then introduced to constrain neighboring timepoints not already linked. This process ensured that the structure of randomly generated STNUs resulted in plan executions that generally flowed from left to right in the plan space, (as is the case with the cooperative rover scenario).

We randomly generated STNUs and tested each for dynamic controllability using the DC algorithm until 100 dispatchable STNUs for each size n were generated. Next, a randomly chosen constraint was loosened by a random number between [1, max_duration = 10]. We then recorded the average runtime of the IDC and DC algorithm to compile the plan. **Fig.11a** indicates that IDC offers up to an order of magnitude speed increase over the DC algorithm on these test cases.

Note that IDC requires additional computation to maintain set of support during the initial compilation using the DC algorithm. **Fig.11b** shows the difference in runtime (up to factor of 3) for the DC algorithm with and without maintaining set of support.

The results indicate that IDC offers a significant speed increase for maintaining dispatchability in real-time, for a comparatively modest cost in precompilation time. The speed of IDC is derived from exploiting the causal structure of a dispatchable plan to efficiently propagate constraint modifications throughout the plan.





(a) (b)

Figure 11: Runtime of (a) Incremental vs. DC Algorithm, (b) DC Algorithm with vs. without maintaining support lists

References

[Bessiere, C. 1994] Arc-consistency and arc-consistency again. *Artificial Intelligence 65*, pages 179–190, 1994.

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

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

[Demestrescu, C., Emiliozzi, S., Italiano, G. 2004] Experimental analysis of dynamic all pairs shortest path algorithms. *Proc.* SODA'04, pp. 362-271.

[Doyle 1979] A truth maintenance system. *AI,* 12:231-272.

[Effinger 2006] Optimal Temporal Planning at Reactive Time Scales via Dynamic Backtracking Branch and Bound, S.M. Thesis, MIT.

[Hofmann, A., Williams, B. 2006] Robust execution of temporally flexible plans for bipedal walking devices. *Proc.* ICAPS-06.

[Koenig, S., Likhachev, M. 2001] Incremental A*. *Advances in Neural Information Processing Systems (14).*

[Morris, P., Muscettola, N. 2000] Execution of temporal plans with uncertainty. In: *Proc.* AAAI-00.

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

[Muscettola, N., Morris, P., Tsmardinos, I. 1998]. Reformulating temporal plans for efficient execution. *Proc. KRR-98*.

[Muscettola, N., Nayak, P., Pell, B., Williams, B. 1998b] To boldly go where no AI system has gone before. *AI* 103(1):5-48.

[Radibeau, G., Knight, R., Chien, S., Fukunaga, A, et al. 1999] Iterative Repair Planning for Spacecraft Operations in the ASPEN System, *Proc.* i-SAIRAS.

[Robertson, P., Williams, B. 2005] A Model-Based System Supporting Automatic Self-Regeneration of Critical Software, *Proceedings of the IFIP/IEEE Intl Workshop on Self-Managed Systems & Services*, France.

[Shu, I., Effinger, R., Williams, C. 2005] Enabling Fast Flexible Planning through Incremental Temporal Reasoning, ICAPS-05.

[Stedl 2004] Managing Temporal Uncertainty Under Limited Communication: A Formal Model of Tight and Loose Team Communication, S.M. Thesis, MIT.

[Stenz 1995] The focused D* algorithm for real-time planning. ICAI.

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

[Vidal, T., Ghallab, M. 1996] Dealing with uncertain durations in temporal constraint networks dedicated to planning. *Proc.* ECAI-96.

[Vidal 1999] Handling contingency in temporal constraint networks: from consistencies to controllabilities. *J. of Exp. & Th. AI*, 11:23-45.

[Vidal 2000]. Controllability characterization and checking in contingent temporal constraint networks. *Proc. KRR-2000.*

[Zweben, M., Davis, E., Daun, B., Deale, M. 1993] Scheduling and rescheduling with iterative repair. *IEEE SMC* 3(6):1588-1596.