EfficientIDC: A Faster Incremental Dynamic Controllability Algorithm

Mikael Nilsson and **Jonas Kvarnström** and **Patrick Doherty**
Department of Computer and Information Science
Linköping University, SE-58183 Linköping, Sweden
{mikni,jonkv,patdo}@ida.liu.se

Abstract

The exact duration of an action generally cannot be predicted in advance. Temporal planning therefore tends to use upper bounds on durations, with the explicit or implicit assumption that if an action happens to be executed more quickly, the plan will still succeed. However, this assumption is often false: If we finish cooking *too early*, the dinner will be cold before everyone is at home and can eat. Simple Temporal Problems with Uncertainty (STPUs) allow us to model such situations. An STPU-based planner must then verify that the networks it generates are executable, captured by the property of *dynamic controllability*. The FastIDC algorithm can do this incrementally during planning. In this paper we show that the FastIDC method can result in traversing part of a temporal network multiple times, with constraints slowly tightening towards their final values. We then present a new algorithm that uses additional analysis together with a different traversal strategy to avoid this behavior. The new algorithm has a guaranteed time complexity lower than that of FastIDC and is proven sound and complete.

Introduction and Background

When planning for multiple agents, for example a joint UAV rescue operation, generating concurrent plans is usually essential. This requires a temporal formalism allowing the planner to reason about the possible times at which plan events will occur during execution. A variety of such formalisms exist in the literature. For example, Simple Temporal Problems (STPs, Dechter, Meiri, and Pearl 1991) allow us to define a set of events related by binary temporal constraints. The beginning and end of each action can then be modeled as an event, and the interval of possible durations for each action as a constraint. However, an STP solution is defined as *any* assignment of timepoints to events satisfying the associated constraints. Thus, if an action is specified to have a duration $d \in [t_1, t_2]$, it is assumed that the planner can *choose* any duration within the interval. In a joint UAV rescue operation this is an unrealistic simplification as there are many actions whose durations cannot be chosen by the planner. Nature is one cause affecting action times, for instance timings of UAV flights and interaction with ground objects will be affected by bad weather.

A formalism which allows uncertainty in timings is STPs with Uncertainty (STPUs) (Vidal and Ghallab 1996), introducing *contingent* constraints, where the time between two events is assumed to be assigned by nature. In essence, if an action is specified to have a contingent duration $d \in [t_1, t_2]$, the "ordinary" constraints must be satisfied for *every* duration within the interval.

In general, STPUs cannot be expected to have static solutions where actions are scheduled at static times in advance. Instead we need dynamic solutions, taking into account the observed times of uncontrollable events (the observed durations of actions). If such a dynamic solution can be found the STPU is *dynamically controllable* (DC) and the plan it represents can be executed correctly regardless of the outcomes of the contingent constraints.

If a plan is *not* dynamically controllable, adding further actions or constraints can never restore controllability. If a planner generates such a plan at some point during search, backtracking will be necessary. To detect this as early as possible the planner should determine after each action is added whether the plan remains DC. For most of the published DC verification algorithms, this would require (re-)testing the entire plan in each step (Morris, Muscettola, and Vidal 2001; Morris and Muscettola 2005; Morris 2006; Stedl 2004). This takes non-trivial time, and one could benefit greatly from using an *incremental* algorithm instead. The only such algorithm in the literature is *FastIDC* (Stedl and Williams 2005; Shah et al. 2007), which was recently proven unsound and corrected (Nilsson, Kvarnström, and Doherty 2013). In this paper we show that the worst-case run-time complexity of FastIDC is at least $O(n^4)$. This worst case can be attained by incrementally changing only one edge in the STPU. We also show that one can use several of the ideas behind FastIDC in a different way, and introduce additional analysis of certain structures, to create a new algorithm with a strictly better amortized run-time of $O(n^3)$.

Definitions

Before we go into details we define the fundamental concepts used in this paper.

Definition 1. *A* ***simple temporal problem (STP)*** *(Dechter, Meiri, and Pearl 1991) consists of a number of real variables $x_1, \ldots, x_n$ and constraints $T_{ij} = [a_{ij}, b_{ij}], i \neq j$ limiting the distance $a_{ij} \leq x_j - x_i \leq b_{ij}$ between the variables.*

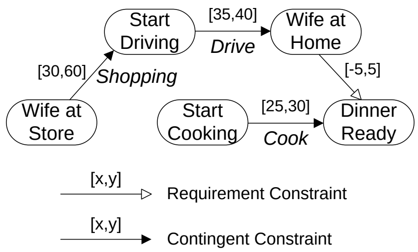

Figure 1: Example STNU.

Definition 2. *A **simple temporal problem with uncertainty (STPU)** (Vidal and Ghallab 1996) consists of a number of real variables $x_1,\ldots,x_n$, divided into two disjoint sets of* controlled timepoints *R and* contingent timepoints *C. An STPU also contains a number of* requirement constraints *$R_{ij} = [a_{ij}, b_{ij}]$ limiting the distance $a_{ij} \le x_j - x_i \le b_{ij}$, and a number of* contingent constraints *$C_{ij} = [c_{ij}, d_{ij}]$ limiting the distance $c_{ij} \le x_j - x_i \le d_{ij}$. For the constraints C_{ij} we require that $x_j \in C$, $0 < c_{ij} < d_{ij} < \infty$.*

We will work with STPs and STPUs in graph form, with timepoints represented as nodes and constraints as labeled edges. They are then referred to as Simple Temporal Networks (STNs) and STNs with Uncertainty (STNUs), respectively. An example is shown in figure 1. In this example a man wants to cook for his wife. He does not want her to wait too long after she returns home, nor does he want the food to wait too long. These two requirements are captured by a single requirement constraint, whereas the uncontrollable (but bounded) durations of shopping, driving home and cooking are captured by the contingent constraints. The question is whether this can be guaranteed regardless of the outcomes of the uncontrollable durations.

Definition 3. *A **dynamic execution strategy** is a strategy for assigning timepoints to controllable events during execution, given that at each timepoint, it is known which contingent events have already occurred. The strategy must ensure that all requirement constraints will be respected regardless of the outcomes for the contingent timepoints.*

*An STNU is **dynamically controllable (DC)** if there exists a dynamic execution strategy for it.*

In figure 1 a dynamic execution strategy is to start cooking 10 time units after receiving a call that the wife starts driving home. This guarantees that cooking is done within the required time, since she will arrive at home 35 to 40 time units after starting to drive and the dinner will be ready within 35 to 40 time units after she started driving.

Any STN can be represented as an equivalent *distance graph* (Dechter, Meiri, and Pearl 1991). Each constraint [u,v] on an edge AB in an STN is represented as two corresponding edges in its distance graph: AB with weight v and BA with weight $-u$. The weight of an edge XY then always represents an upper bound on the temporal distance from its source to its target: $time(Y) - time(X) \le weight(XY)$. Computing the all-pairs-shortest-path (APSP) distances in the distance graph yields a *minimal representation* containing the tightest distance constraints that are implicit in the original problem (Dechter, Meiri, and Pearl 1991). This directly corresponds to the tightest interval constraints $[u', v']$ implicit in the original STN. If there is a negative cycle in the distance graph, then no assignment of timepoints to variables satisfies the STN: It is *inconsistent*.

In the same way, an STNU always has an equivalent *extended distance graph* (Stedl 2004).

Definition 4. *An **extended distance graph (EDG)** is a directed multi-graph with weighted edges of 5 kinds: positive requirement, negative requirement, positive contingent, negative contingent and conditional.*

Requirement edges and contingent edges in an STNU are translated into pairs of edges of the corresponding type in a manner similar to what was previously described for STNs. A conditional edge (Stedl 2004) is never present in the initial EDG, but can be derived from other constraints through calculations discussed in the following section (figure 2).

Definition 5. *A **conditional edge** CA annotated $< B, -w >$ encodes a conditional constraint: C must execute after B* or *at least w time units after A, whichever comes first. The node B is called the* conditioning node *of the constraint/edge.*

The *FastIDC* Algorithm

The FastIDC algorithm (Stedl and Williams 2005; Shah et al. 2007) incrementally checks the dynamic controllability of an STNU. While it can be used for loosening constraints, we will focus only on the part responsible for adding/tightening constraints. The reason is that when it is detected that a plan is not DC, a planner would normally backtrack to a previously visited plan (whose associated STNU can be saved), which does not require arbitrary loosening of existing constraints.

FastIDC (algorithm 1) is based on a set of *derivation rules* (figure 2). When a constraint is added or tightened in an STNU, FastIDC will derive the effects of this on existing events. It does so by making implicit constraints explicit, adding them to the STNU. The new constraints may then lead to more derivations and so the process propagates until no more constraints can be derived.

The version of FastIDC shown here is slightly modified compared to the original presentation. First, it includes an adaptation of the soundness fix described in Nilsson, Kvarnström, and Doherty (2013) which checks for cycles containing only negative edges. Second, the presentation has been streamlined by assigning rule IDs (D8 and D9 in figure 2) to the general and unordered reduction rules (Morris, Muscettola, and Vidal 2001) which are required for soundness. In figure 2 all variables are assumed to be positive, i.e. '-v' is negative, with the exception of '-u' in D8 which may be either negative or positive.

Being incremental, FastIDC assumes that at some point a dynamically controllable STNU was already constructed (for example, the empty STNU is trivially DC). Now one or more requirement edges $e_1,\ldots,e_n$ have been added or tightened, together with zero or more contingent edges and zero or more new nodes, resulting in the graph G. FastIDC should then determine whether G is DC.

Algorithm 1: FastIDC – sound version

function FAST-IDC($G, e_1, \ldots, e_n$)
Q ← sort $e_1, \ldots, e_n$ by distance to temporal reference
  (order important for efficiency, irrelevant for correctness)
for *each modified edge e_i in ordered Q* **do**
  if IS-POS-LOOP(e_i) **then** SKIP e_i
  if IS-NEG-LOOP(e_i) **then return** false
  for *each rule (Figure 2) applicable with e_i as focus* **do**
    if *edge z_i in G is modified or created* **then**
      Update *CCGraph*
      if *Negative cycle created in CCGraph* **then**
      return false
      if *G is squeezed* **then return** false
      if not FAST-IDC(G, z_i) **then return** false
    end
  end
end
return true

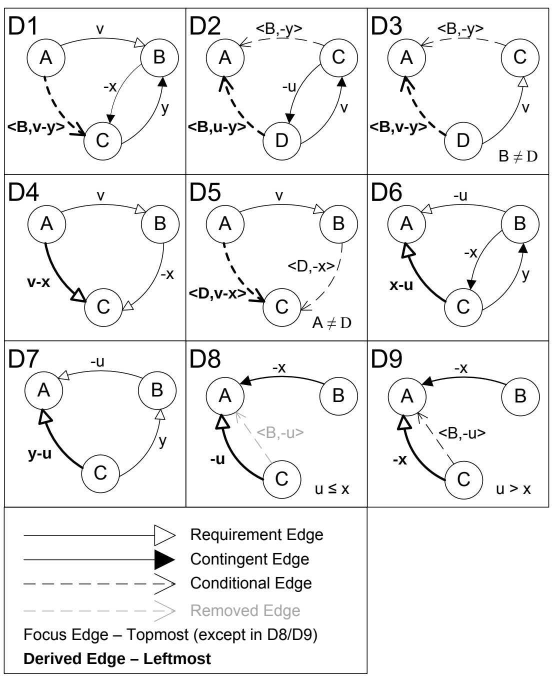

Figure 2: FastIDC Derivation Rules.

The algorithm works in the EDG of the STNU. First it adds the newly modified/added requirement edges to a queue, Q (a contingent edge must be added before any other constraint is added to its target node and is then handled implicitly through requirement edges). Q is sorted in order of decreasing distance to the *temporal reference* (*TR*), a node always executed before all other nodes at time zero. Therefore nodes close to the "end" of the STNU will be dequeued before nodes closer to the "start". This to some extent prevents duplication of effort by the algorithm, but is not essential for correctness or for understanding the derivation process.

In each iteration an edge e_i is dequeued from Q.

A positive loop (an edge of positive weight from a node to itself) represents a trivially satisfied constraint that can be skipped. A negative loop entails that a node must be executed before itself, which violates DC and is reported.

If e_i is not a loop, FastIDC determines whether one or more of the derivation rules in figure 2 can be applied with e_i as focus. The topmost edge in the figure is the focus in all rules except D8 and D9, where the focus is the conditional edge $< B, -u >$. Note that rule D8 is special: The derived requirement edge represents a stronger constraint than the conditional focus edge, so the conditional edge is removed.

For example, consider rule D1. This rule will be matched if e_i is a positive requirement edge, there is a negative contingent edge from its target B to some other node C, and there is a positive contingent edge from C to B. Then a new constraint (the bold edge) can be derived. This constraint is only added to the EDG if it is strictly tighter than any existing constraint of the same type between the same nodes.

More intuitively, D1 represents the situation where an action is started at C and ends at B, with an uncontrollable duration in the interval $[x, y]$. The focus edge AB represents the fact that B, the end of the action, must not occur more than v time units after A. This can be represented more explicitly with a conditional constraint AC labeled $< B, v-y >$: If B has occurred (the action has ended), it is safe to execute A. If at most $v-y$ time units remain until C (equivalently, at least $y-v$ time units have passed *after* C), no more than v time units can remain until B occurs, so it is also safe to execute A.

Whenever a new edge is created, the corrected FastIDC tests whether a cycle containing only negative edges is generated. The test is performed by keeping the nodes in an incrementally updated topological order relative to negative edges. The unlabeled graph which is used for keeping the topological order is called the *CCGraph*. It contains the same nodes as the EDG and has an edge between two nodes iff there is a negative edge between them in the EDG.

After this a check is done to see if the new edge *squeezes* a contingent constraint. Suppose FastIDC derives a requirement edge BA of weight w, for example $w = -12$, representing the fact that B must occur at least 12 time units after A. Suppose there is also a contingent edge BA of weight $w' > w$, for example $w' = -10$, representing the fact that an action started at A and ending at B may in fact take as little as 10 time units to execute. Then there are situations where nature may violate the requirement edge constraint, and the STNU is not DC. Situations where a meeting a requirement constraint would force a contingent constraint to be squeezed

If the tests are passed and the edge is tighter than any existing edges in the same position, FastIDC is called recursively to take care of any derivations caused by this new edge. Although perhaps not easy to see on a first glance, all derivations lead to new edges that are closer to the temporal reference. Derivations therefore have a direction and will eventually stop. When no more derivations can be done the algorithm returns true to testify that the STNU is DC.

Analysis of FastIDC

We will now analyze certain aspects of the FastIDC algorithm in more detail.

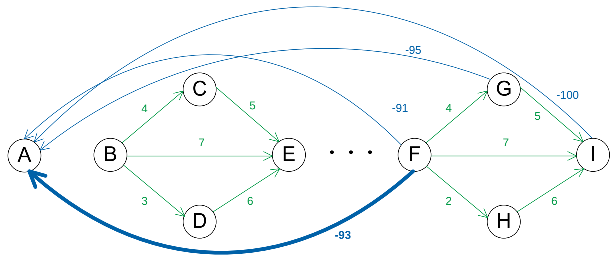

Figure 3: Why depth first is a suboptimal strategy.

Edge processing order. Following (Shah et al. 2007), the initial list of modified edges is processed in order of distance to the temporal reference, but all edges derived by FastIDC itself are handled recursively and depth-first. The small example in figure 3 shows why this is a suboptimal strategy for selecting focus edges. In this example the positive edges are present in the initial graph. The negative IA edge is added as the only edge in this increment. Thus FastIDC is called and Q will contain only $e_1 = IA$. This leads to derivation of the GA edge, with weight -95. The depth first strategy then derives FA, weight -91, and additional edges moving toward the start of the STNU. However at a later step the IA and FI edges will be used to derive a strictly smaller weight for FA, -93. This derivation will then propagate to decrease the weights of all the previously derived edges. In the worst case a number of paths of positive edges from A to I, proportional to the total number of paths, may be traversed in reverse by FastIDC as new negative weights are incrementally derived. There is an exponential number of different paths in a graph which makes this worst case suboptimal.

An Improved Search Strategy. As noted above, the algorithm as published sorts the initial list of modified edges but processes newly derived edges depth first. This can be improved by keeping a *global* priority queue Q of modified edges. When a new edge is derived, it is not processed recursively but added at the proper place in this queue. The algorithm then iterates until the queue of modified edges is empty. The effect is that in each iteration the algorithm chooses among *all* known modified edges the one that is the furthest from the temporal reference, as was perhaps intended by the authors but not realized in the pseudo-code.

A Lower Bound on Time Complexity. An example will now show that even with the improved search strategy, the worst case run-time complexity of FastIDC is still at least $O(n^4)$ when processing the tightening of one edge.

The left part of figure 4 shows a part of an EDG created by FastIDC when incrementally adding constraints to an STNU. The figure contains three categories of nodes: A, B and C nodes. All B nodes are connected in sequence by edges of weight 1 as illustrated in the figure. So are the A nodes, except the one with highest index. A_0 is connected to all B nodes by edges whose weights increase with the indices of B nodes. There is also one edge of weight 100 from each B node to each A node. These $|A| \cdot |B|$ edges are omitted in the figure for clarity.

The nodes in the figure are ordered from left to right by

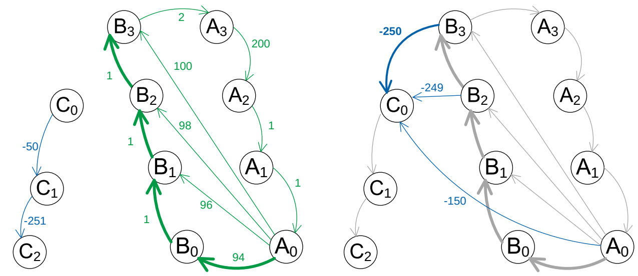

Figure 4: High complexity scenario part 1.

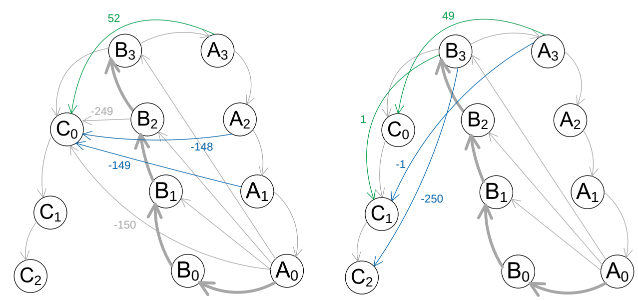

Figure 5: High complexity scenario part 2.

path distance to the temporal reference node (TR), which is not shown in the figures. This means that there are negative edges or paths from the nodes to the TR and that these are more negative the further to the right in the figure a node is placed. Recall that negative edges are sorted in the FastIDC queue by the distance from their source node to the TR.

FastIDC derives edges by giving higher priority to negative edges whose source nodes are closer to the end of the EDG. The example in figure 4 contains a shortest path A_0, B_0, B_1, B_2, B_3 from the end towards earlier nodes. However this order works against FastIDC since derivation rule D7 derives tighter constraints in the *opposite* direction (the source of the derived edge is that of the positive edge). We will exemplify this now.

Suppose the $B_3 \rightarrow C_0$ edge shown in bold in the right part of figure 4 is added or tightened to a weight of -250 and that FastIDC is called with this edge as e_1. FastIDC will find two applicable derivations where this is the focus edge. Both derivations are instances of D7, resulting in $B_2 \rightarrow C_0$ and $A_0 \rightarrow C_0$ being created and placed in the queue.

In the next iteration, $A_0 \rightarrow C_0$ has the highest priority (because A_0 is farther from the TR than B_2 is), and will be taken from the queue. When processing this edge, derivations using D7 will combine it with the "hidden" edges $B_i \rightarrow A_0$ with weight 100 to derive $|B|$ edges $B_i \rightarrow C_0$. These are all put in the queue. An edge $A_1 \rightarrow C_0$ with weight -149 will also be generated and ends up in front of the queue (figure 5).

When $A_1 \rightarrow C_0$ is taken for processing, $|B|$ new edges are derived through combination with $B_i \rightarrow A_1$, but these are discarded since they have higher weights than the previously derived edges in their positions. An edge $A_2 \rightarrow C_0$ with weight -148 is also derived and will be processed first.

Processing this leads to a similar procedure, again with $|B|$ edges discarded. The edge $A_3 \to C_0$ is among those derived. Since it has a positive weight it ends up last in the queue (sorted on the distance from C_0 to the TR). The left part of figure 5 shows the current situation.

At this point the edges from $B_i \to C_0$ will be taken from the queue and not lead to any derivations until $B_2 \to C_0$ is processed. This is used to derive tighter $B_1 \to C_0$ and $A_0 \to C_0$ edges which in turn follow the pattern just described leading to tightenings of the $A_i \to C_0$ edges. This happens again when $B_0 \to C_0$ and $A_0 \to C_0$ are tightened. At this point the $A_3 \to C_0$ edge reaches its final weight 49. The queue is then processed until $A_3 \to C_0$ is removed as the last edge in the queue. This leads to derivation of $A_3 \to C_1$ with weight -1 which in turn leads to $B_3 \to C_1$ of weight 1 and $B_3 \to C_2$ with weight -250. Here we again have an edge from B_3 with weight -250. FastIDC will then continue the exact same sequence as before but now deriving edges toward C_2 instead of C_0.

It is possible that $|A|$, $|B|$ and $|C|$ are all $O(n)$ and follow the same pattern as in the example. Then there are $O(n)$ spins around the $A-B$ cycle as the target of the negative $B \to C$ edges traverses all C nodes. Each spin around the cycle takes $O(n^3)$ time: There are $O(n)$ updates to $A_0 \to C_x$ and each of these updates the $O(n)$ $A_i \to C_x$ edges, each of which tries to update the $O(n)$ $B \to C_x$ edges. The worst case complexity of one call to FastIDC must therefore be at least $O(n^4)$.

Unfortunately, even though the structure in the example requires the addition of many edges that are handled quickly by FastIDC, the complexity cannot be amortized to reach a lower value. The problem is that FastIDC will always pay the full $O(n^4)$ price each time the $B_3 \to C_0$ edge in the example is tightened. This may happen as part of other tightenings or by direct change many times as the final STNU is built. As such there are no cheaper increments that can pay for the more expensive ones.

One cause for this complexity is the existence of a region of nodes (the A and B nodes) where there is at least initially no forced ordering between the nodes. We will now present a new way of handling such regions.

The *EfficientIDC* Algorithm

We now present the Efficient Incremental Dynamic Controllability checking algorithm (Algorithm 2, *EfficientIDC* or *EIDC* for short). The key to EIDC's efficiency is the use of focus *nodes* instead of focus edges. When EIDC tightens an edge, it adds the target of this edge as a new focus node to be processed. When EIDC processes a focus node n, it applies all derivation rules that have an incoming edge to n as focus edge, guaranteeing that no tightenings are missed.

The use of a focus node allows us to use a modified version of Dijkstra's algorithm to efficiently process parts of an EDG in a way that avoids certain forms of repetitive intermediate edge tightenings performed by FastIDC. The key to understanding this is that derivation rules essentially calculate shortest distances. For example, rule D4 states that if we have tightened edge AB and there is an edge BC, an edge AC may have to be tightened to indicate the length of the shortest path between A and C. Shortest path algorithms cannot be applied indiscriminately, since there are complex interactions between the different kinds of edges, but can still be applied in certain important cases.

Algorithm 2: The EfficientIDC Algorithm

```
function EfficientIDC(EDG G, DDG D, CCGraph C, edge e)
todo ← {Target(e)}
if e is negative and e ∉ C then
    add e to C
    if negative cycle detected then return false
    todo ← todo ∪ {Source(e)}
end
while todo ≠ ∅ do
    current ← pop some n from todo where
        ∀e ∈ Incoming(C,n) : Source(e) ∉ todo
    ProcessCond(G,D,current)
    ProcessNegReq(G,D,C,current)
    ProcessPosReq(G,current)
    for each edge e added to G in this iteration do
        if Target (e) ≠ current then
            todo ← todo ∪{Target(e)}
        end
        if e is a negative requirement edge and e ∉ C then
            add e to C
            if negative cycle detected then return false
            todo ← todo ∪{Target(e), Source(e)}
        end
    end
    if G is squeezed then return false
end
return true
```

The *final* tightening performed for each edge will still be identical in EIDC and FastIDC, which is required for correctness. An extensive example will be provided below.

As in (the corrected) FastIDC, the EDG is associated with a *CCGraph* used for detecting cycles of negative edges. The graph also helps EIDC determine in which order to process nodes: In reverse temporal order, from the "end" towards the "start", taking care of incoming edges to one node in each iteration. The EDG is also associated with a *Dijkstra Distance Graph* (*DDG*), a new structure used for the modified Dijkstra algorithm as described below. To simplify the presentation, EIDC will be given one new or tightened requirement edge e at a time.

The EfficientIDC algorithm. First, the target of e is added to *todo*, a set of focus nodes to be processed.

If e is a *negative* requirement edge, a corresponding edge is added to the *CCGraph C* which keeps track of all negative edges. If this causes a negative cycle, G is not DC. Else, the source of e should also be processed for efficiency, as this may produce new edges into Target(e), and is added to *todo*.

Iteration. As long as there are nodes to process:

A node to process, *current*, is selected and removed from *todo*. Incoming negative edges e to the chosen node n must not originate in a node also marked as *todo*: Otherwise, Source(e) should be processed first, since this has the potential of adding new incoming edges to n. There is always a *todo* node satisfying this criterion, or there would be a cycle of negative edges which would have been detected.

Then it is time to process all existing incoming edges.

Incoming conditional edges are processed as FastIDC focus edges using ProcessCond(). This function is equivalent to applying rules D2, D3, D8 and D9, but does so for a larger part of the graph in a single step.

There are only $O(n)$ contingent constraints in an EDG and hence only $O(n)$ conditioning nodes (which have to be the target of a contingent constraint). All times in conditional constraints/edges are measured towards the source of the contingent constraint. This means that all conditional constraints conditioned on the same node have the same target.

Now, it is important to note that EIDC processes conditional edges conditioned on the same node separately. This is possible because FastIDC does not "mix" conditional edges with different conditioning nodes in any of the rules, so they cannot be derived "from each other".

For any condition node c, the function finds all edges that are conditioned on c and have *current* as target. We now in essence want to create a single destination shortest path tree rooted in *current*. We can do this using Dijkstra's algorithm given that edges are added to the DDG in the reverse direction and given that no edge weights are negative. To achieve the latter we let *minw* be the absolute value of the most negative edge weight and add this weight to all conditional edges. Then Dijkstra's algorithm is run but allowed to stop in any direction as soon as the *minw* distance is reached. This will in a single call to Dijkstra derive a final set of shortest distances that FastIDC might have had to perform a large number of iterations to converge towards.

The function directly applies the "special" derivation rules D8 and D9, which convert conditional edges to requirement edges, to the result. It then checks whether any calculated shortest distance corresponds to a new derived edge, corresponding to applying D2 and D3 over the processed part of the graph. Note: If a conditional edge is derived and reduced by D8 rather than D9, it will cause a negative requirement edge to also be added for a total of two new edges.

This function may generate new incoming requirement edges for *current*, which is why it must be called before incoming requirement edges are processed.

Incoming negative requirement edges are processed using ProcessNegReq(). This function is almost identical to *ProcessCond* with the only differences being that the edges are negative requirement instead of conditional and that because of this there is no need to apply the D8 and D9 derivations. Applying the calculated shortest distances in this case corresponds to applying the derivation rules D6 and D7.

This function may generate new incoming positive requirement edges for *current*, which is why it must be called before incoming requirement edges are processed.

Incoming positive requirement edges are processed using ProcessPosReq(), which applies rules D1, D4 and D5.

These are the only possible types of focus edge, and therefore all focus edges that could possibly have given rise to the *current* focus node have now been processed.

We now check all edges that were derived above. Edges that do not have *current* as a target need to be processed, so their targets are added to *todo*. If there is a negative requirement edge that is not already in the *CCGraph*, this edge represents a new forced ordering between two nodes. We must then update the *CCGraph* and check for negative cycles. If a new edge is added to the *CCGraph* both the source and the target of the edge must be added to *todo*.

Finally, EIDC verifies that there is no squeeze when a new edge is added, precisely as FastIDC does.

Updating the CCGraph. By letting the *CCGraph* contain the transitive closure of its edges we enable the algorithm to select nodes for processing in the best known order. As will be seen later this has a direct impact on its run-time.

Updating the DDG graph. The DDG graph contains weights and directions of edges that FastIDC derivations use to derive new edges, and is needed to process edges effectively. The DDG contains:

1. The positive requirement edges of the EDG, in reverse direction
2. The negative contingent edges of the EDG, with weights replaced by their absolute values

To make the algorithm easier to read, updates have been omitted. Updating the DDG is quite simple. When a positive edge is added to the EDG it is added to the DDG in reversed direction. Negative contingent edges also have to be added to the DDG. In case a positive requirement edge disappears from the EDG it is removed from the DDG.

Example

We now go through a detailed example of how EIDC processes the three kinds of incoming edges. Like before, dashed edges represent conditional constraints, filled arrowheads represent contingent constraints, and solid lines with unfilled arrowheads represent requirement constraints.

Figure 6 shows an initial EDG constructed by incrementally calling EIDC with one new edge at a time. We will initially focus on the nodes and edges marked in black, while the gray part will be discussed at a later stage.
Assume we add a new requirement edge of weight -10 as shown in the rightmost part of figure 7. When we call EIDC for this edge, *current* will become the target of this edge, X. No incoming conditional edges exist, but there is now one incoming negative requirement edge to be processed. Using Dijkstra's algorithm we generate two requirement edges of weight 25 and 30 that also end in X. Now *current* has two incoming positive requirement edges that are also processed

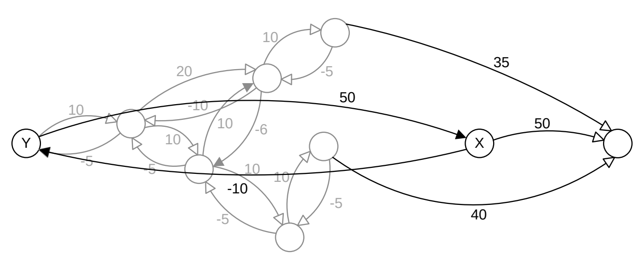

Figure 6: Initial EDG.

Algorithm 3: Helper Functions

function ProcessCond(*G,D,current*)
$allcond \leftarrow$ IncomingCond$(current, G)$
$condnodes \leftarrow \{n \in G \mid n$ is the conditioning node of some $e \in allcond\}$
for *each* $c \in condnodes$ **do**
  $edges \leftarrow \{e \in allcond \mid$ conditioning node of e is $c\}$
  $minw \leftarrow |min\{weight(e) : e \in edges)\}|$
  add *minw* to the weight of all $e \in edges$
  for $e \in edges$ **do**
    add e to D with reversed direction
  end
  LimitedDijkstra (current, D, *minw*)
  for *all nodes n reached by* LimitedDijkstra **do**
    $e \leftarrow$ cond. edge $(n \rightarrow current)$, weight Dist (n) - *minw*
    if *e is a tightening* **then**
      add e to G
      apply D8 and D9 to e
    end
  end
  Revert all changes to D
end
return

function ProcessNegReq(*G,D,current*)
$edges \leftarrow$ IncomingNegReq$(current, G)$
$minw \leftarrow |min\{weight(e) : e \in edges)\}|$
add *minw* to the weight of all $e \in edges$
for $e \in edges$ **do**
  add e to D with reversed direction
end
LimitedDijkstra (current, D, *minw*)
for *all nodes n reached by* LimitedDijkstra **do**
  $e \leftarrow$ req. edge $(n \rightarrow current)$ of weight Dist (n) - *minw*
  if *e is a tightening* **then** add e to G
end
Revert all changes to D
return

function ProcessPosReq(*G,current*)
for *each* $e \in$ IncomingPosReq$(current, G)$ **do**
  apply derivation rule D1, D4 and D5 with e as focus edge
end
return

(using D1) to generate two conditional edges having Y as target. Since these are not incoming to X they require no further processing at the moment. However, the addition of the edges leads to Y being marked as *todo*.

In the next iteration, Y is selected as *current*. We now focus on the part of the EDG shown in figure 8 and see that Y has two incoming conditional edges with the same conditioning node X. These edges are processed together, resulting in a *minw* value of 25. After adding reversed conditional edges with this weight increase, we have the DDG used for Dijkstra as shown in figure 9. In the figure we have labeled each node with its shortest distance from Y in the DDG.

Processing $current = Y$ gives rise to the bold edges in figure 10. We consider how the -9 edge is created. First the distance to the source node of the -9 edge is calculated by

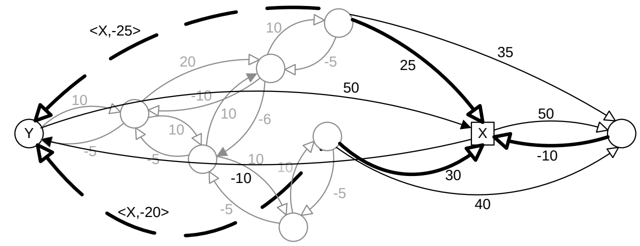

Figure 7: Derivation of smaller scenario.

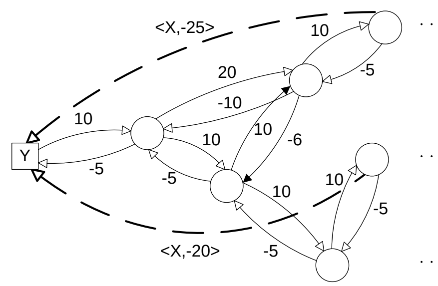

Figure 8: Example scenario for conditional edges.

Dijkstra's algorithm. This is 16 (see figure 9). Subtraction of 25 gives a conditional edge with weight -9. However, since the lower bound of the contingent constraint involving X is 10, D8 is then applied to remove the conditional edge and create a requirement edge with weight -9. The distance calculation corresponds in this case to what FastIDC would derive by applying first D3 and then D6 with the conditional $< X, -25 >$ edge as focus.

The example shows that we need to add *minw* to get positive edges for Dijkstra's algorithm to work with. Note that all new edges need to be checked for local consistency and all negative edges should be added to the cycle checking graph.

Correctness of the *EfficientIDC* Algorithm

The following theorem states the correctness of the algorithm based on the corrected version of FastIDC as proven by Nilsson, Kvarnström, and Doherty (2014). Since FastIDC may update the same edge more than once when processing an increment the theorem compares the EDG of EfficientIDC against the final result of FastIDC. In the proof it is

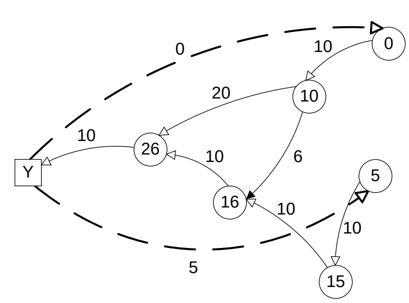

Figure 9: Dijkstra Distance Graph of the small scenario.

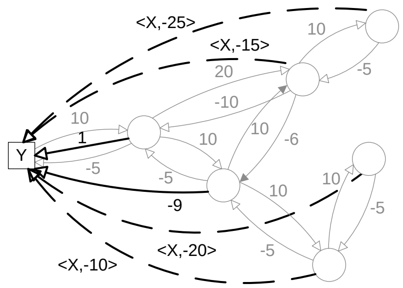

Figure 10: Result of processing $current = Y$.

assumed for simplicity that each call consists of only one edge tightening/addition. Any increment consisting of more changes can be broken down to several without affecting the end results, thus preserving correctness.

Theorem 1. *Let G be an EDG of a DC STNU and e be a single tightened edge in G. Let G' be the graph produced by FastIDC(G,e) and let G'' be the graph produced by EfficientIDC(G,e). Then $G' = G''$. Additionally, the algorithms agree on whether the corresponding STNU is dynamically controllable.*

Proof. (Sketch) First, derivation rules only generate sound conclusions. The derivations performed by EIDC are either through direct use of derivation rules or through the use of Dijkstra in a way that corresponds directly to repeated application of derivation rules. Therefore EIDC is sound in terms of edge generation.

Second, completeness requires that for every tightened edge, all applicable derivation rules are applied. When an edge is tightened, EIDC always adds the target node to *todo*. All nodes in *todo* will eventually be processed, and when a node *current* is removed from *todo*, all derivation rules applicable with any incoming edge as focus are applied. This is guaranteed since the last time a node is processed as current all nodes that will be executed after it have been processed and it is only via these that new incoming edges can be derived. Since all these nodes have had all derivation rules applied to them so becomes the case also for current. Applying the rules is done either directly or indirectly through the use of Dijkstra's algorithm. Therefore no derivations can be missed and EIDC is complete in terms of edge generation.

Thus, the algorithms eventually derive the same edges. Since they both check the DC property in the same way they also agree on which STNUs are DC and which are not. □

Run-time Complexity of *EfficientIDC*

Since the run-time of EfficientIDC is the main focus of this paper we formulate it as a theorem.

Theorem 2. *The run-time of EfficientIDC when processing one tightened or added edge is $O(n^4)$ in worst case but $O(n^3)$ amortized over the creation of an STNU, where n is the number of nodes.*

Proof. When EIDC adds a negative requirement edge e, it checks whether this is already represented in the CCGraph. If not ($e \notin C$), the edge previously had positive weight, and its new negative weight represents a new forced ordering.

First, assume this does *not* happen: Whenever a new negative requirement edge e is created, it is already in C.

Consider what happens when a node X is selected and removed from *current*. Clearly X cannot have incoming negative edges from any node in *todo*. After it is processed, the only way that X could enter *todo* again is (given our assumption) as the *target* of an edge derived from *another* node Y. Such edges are only created in ProcessPosReq(), and must therefore have been created using derivation rules D1, D4 or D5. We see in figure 2 that for these to be applicable, X must have an incoming negative edge from Y (possibly alongside a contingent or conditional edge). Then Y would have to be executed before X due to the negative weight edge.

Clearly Y was not in *todo* when X was selected, or it would have been selected before X. It must have been added to *todo* later. Then there must be a chain of nodes responsible for the addition of Y into *todo*. Like every chain, this chain must originate in some node Z that was in *todo* when X was selected.

We know $Z \neq X$, since X cannot lead to itself later being added to *todo*: Then it must be "before itself", since only *later* nodes can derive new edges into a node (excepting the fact that a node can itself derive new incoming edges).

Also, Z cannot be a node that is executed after X, since then X would not be chosen. But since Y is added to *todo* as a consequence of derivations via Z, we know that Z is executed after Y. And since Y is executed after X we now see that Z must be executed after X. Therefore there is a negative edge between them and Z cannot have been in *todo* when X was executed. This leads to a contradiction.

Thus, X cannot be added to *todo* after it has been removed. As a consequence each node is only processed once, assuming no new order is detected (no new edge added to C). We now analyze the complexity of EIDC under this assumption.

Complexity 1. Since each node can be selected as *current* at most once, the main **while** loop iterates $O(n)$ times.

In each iteration, the incoming positive requirement edges for *current* can be processed in $O(n^2)$ time: Each derivation is $O(1)$ and there are at most $O(n)$ incoming positive edges which can find at most $O(n)$ outgoing edges for derivations.

Processing incoming conditional and negative requirement edges is more complicated, due to the use of Dijkstra's algorithm. Conditional edges require slightly more work than negative requirement edges and as such provide an upper bound for both types. The cost of updating the DDG used for Dijkstra calculations is $O(1)$ per edge change which is hidden in the normal cost of adding edges. The following list shows the complexity of the different steps done when processing conditional edges conditioned on *one* node.

1. Add conditional edges to the DDG, $O(n)$
2. Find *minw* among these, $O(n)$
3. Replace weights on the negative contingent edges, $O(n)$
4. Run the limited Dijkstra's algorithm $O(n^2)$
5. Add new conditional/requirement edges to the EDG, $O(n)$
6. Remove conditional edges from the DDG, $O(n)$

This sums to $O(n^2)$ for processing all conditional edges conditioned on *one* node. Taking care of all conditioning nodes throughout the EDG causes the procedure to be carried out $O(n)$ times and incurs an $O(n^3)$ aggregated cost.

It follows from the described procedure that processing negative requirement edges for *current* takes $O(n^2)$ time.

Each outer loop adds $O(n^2)$ new edges. Checking local consistency takes $O(n^2)$ time. Adding them to the *CCGraph* takes accumulated $O(n^3)$ time over the whole increment.

The final step is to choose the next *current* node for processing and this is done by picking any node from *todo* that has no predecessors in *todo*. In practice a list of candidates is kept which is updated every time a node has been removed from *todo*. This is clearly below $O(n^2)$ and done once in each outer iteration, for a total below $O(n^3)$.

Second, we consider what happens when new orderings are found and added to C while processing an increment.

Let X be the node that was found to be ordered after *current*. Finding all incoming edges to *current* depends on the fact that all nodes ordered after it, including X, must have been processed before *current*. If X was processed after *current*, edges targeting *current* that could be derived via X may be missed and the algorithm would not be complete. These are however the only edges targeting *current* that would be missed. So the algorithm goes back to process X and then reprocesses *current* to find these edges.

An order such as the one just discovered can only be found when processing a node that is ordered after *current* or when processing *current*. The new edge must be derived through interaction of a positive edge and a negative edge targeting *current*, i.e. it would be found at *current* or when processing the source of the negative edge which is by definition ordered after *current*. If the new order is found when processing any other node ordered after *current* there is no need for reprocessing as the requirement for finding all edges at *current* is satisfied.

Complexity 2. New orderings that lead to reprocessing of nodes are detected when the node needing reprocessing is being processed as *current*. As such the cost for reprocessing is only that of one iteration in the algorithm per new ordering found. Over the course of constructing an STNU there can be $O(n^2)$ new orderings found, however each may only affect the same node $O(n)$ times. This is important since the cost of processing a node is $O(n^2)$ plus any processing of conditional edges for up to a total of $O(n^3)$, for instance if this node is the target of a maximum of conditional edges in the STNU. Regardless of how the cost of processing conditional nodes is spread throughout the STNU they may be involved in reprocessing $O(n)$ times in the worst case. This leads to an $O(n^4)$ bound on building the whole STNU including any reprocessings.

The worst case is therefore $O(n^4)$ for one increment. However, the amortized work done when reprocessing stays at $O(n^2)$ per increment for a total amortized time of $O(n^3)$, taking into consideration that without reprocessing an increment may still take $O(n^3)$. □

Conclusion

A new way of incrementally testing dynamic controllability is presented. It is more efficient than FastIDC but provides the same result, both in form of EDG and DC classification. Higher efficiency is gained by observing that FastIDC is inefficient when deriving constraints over unordered sections in the EDG. EIDC overcomes this by applying Dijkstra's algorithm to quickly derive all constraints over such sections. The EDG processed by EIDC is dispatchable since it derives the same constraints as FastIDC.

Acknowledgments

This work is partially supported by the Swedish Research Council (VR) Linnaeus Center CADICS, the ELLIIT network organization for Information and Communication Technology, the Swedish Foundation for Strategic Research (CUAS Project), the EU FP7 project SHERPA (grant agreement 600958), and Vinnova NFFP6 Project 2013-01206.

References

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

Morris, P., and Muscettola, N. 2005. Temporal dynamic controllability revisited. In *Proceedings of the 20th National Conference on Artificial Intelligence (AAAI)*, 1193–1198. AAAI Press / The MIT Press.

Morris, P.; Muscettola, N.; and Vidal, T. 2001. Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, 494–499. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Morris, P. 2006. A structural characterization of temporal dynamic controllability. In *Proceedings of the 12th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 4204 of *Lecture Notes in Computer Science*, 375–389. Springer.

Nilsson, M.; Kvarnström, J.; and Doherty, P. 2013. Incremental dynamic controllability revisited. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS)*.

Nilsson, M.; Kvarnström, J.; and Doherty, P. 2014. Classical Dynamic Controllability Revisited: A Tighter Bound on the Classical Algorithm. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence (ICAART)*.

Shah, J. A.; Stedl, J.; Williams, B. C.; and Robertson, P. 2007. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In Boddy, M. S.; Fox, M.; and Thibaux, S., eds., *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, 296–303. AAAI Press.

Stedl, J., and Williams, B. 2005. A fast incremental dynamic controllability algorithm. In *Proceedings of the ICAPS Workshop on Plan Execution: A Reality Check*.

Stedl, J. L. 2004. Managing temporal uncertainty under limited communication: A formal model of tight and loose team coordination. Master's thesis, Massachusetts Institute of Technology.

Vidal, T., and Ghallab, M. 1996. Dealing with uncertain durations in temporal constraints networks dedicated to planning. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI)*, 48–52.