Efficient Processing of Simple Temporal Networks with Uncertainty: Algorithms for Dynamic Controllability Verification

Mikael Nilsson, Jonas Kvarnström and Patrick Doherty

Noname manuscript No.
(will be inserted by the editor)

Efficient Processing of Simple Temporal Networks with Uncertainty

Algorithms for Dynamic Controllability Verification

Mikael Nilsson · Jonas Kvarnström · Patrick Doherty

Received: date / Accepted: date

Abstract Temporal formalisms are essential for reasoning about actions that are carried out over time. The exact durations of such actions are generally hard to predict. In temporal planning, the resulting uncertainty is often worked around by only considering upper bounds on durations, with the assumption that when 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 ready to eat.

Using *Simple Temporal Networks with Uncertainty (STNU)*, a planner can correctly take both lower and upper duration bounds into account. It must then verify that the plans it generates are executable regardless of the actual outcomes of the uncertain durations. This is captured by the property of *dynamic controllability* (DC), which should be verified incrementally during plan generation.

Recently a new incremental algorithm for verifying dynamic controllability was proposed: EfficientIDC, which can verify if an STNU that is DC remains DC after the addition or tightening of a constraint (corresponding to a new action being added to a plan). The algorithm was shown to have a worst case complexity of $O(n^4)$ for each addition or tightening. This can be amortized over the construction of a whole STNU for an amortized complexity in $O(n^3)$.

In this paper we improve the EfficientIDC algorithm in a way that prevents it from having to reprocess nodes. This improvement leads to a lower worst case complexity in $O(n^3)$.

Keywords Simple Temporal Networks with Uncertainty · Dynamic Controllability · Incremental Algorithm

This paper is based on an earlier paper at TIME-2014 [13]

M. Nilsson · J. Kvarnström · P. Doherty
Department of Computer and Information Science Linköping University
SE-58183 Linköping, Sweden
E-mail: {mikni,jonkv,patdo}@ida.liu.se

1 Introduction and Background

When planning for multiple agents, for example a joint Unmanned Aerial Vehicle (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 exists in the literature. For example, Simple Temporal Networks (STNs [4]) 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 can be modeled as a constraint related to the action's start and end event: $dur = end - start \in [min, max]$.

However, an STN solution is defined as *any* assignment of timepoints to events that satisfies the associated constraints. When an action has a duration $dur \in [min, max]$, it is sufficient that the remaining constraints can be satisfied for *some* duration within this interval. This corresponds to the case where the planner can freely *choose* action durations within given bounds, which is generally unrealistic. For example, nature can affect action durations: timings of UAV flights and interactions with ground objects will be affected by weather and wind.

A formalism allowing us to model durations that we cannot directly control is STNs with Uncertainty (STNUs) [17]. This formalism introduces *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 other constraints must be satisfiable for *every* duration that nature might assign within the given interval.

All constraints modeled in STNs and STNUs are binary. Because of this we can also model any STN(U) as an equivalent graph, where each constraint is represented by a labeled edge and each event by a node.

Example 1 Suppose that a man wants to surprise his wife with some nice cooked food after she returns from shopping. For the surprise to be pleasant he does not want her to have to wait too long for the meal after returning home. He also does not want to finish cooking the meal too early so it has to lay waiting. We can model this scenario with an STNU as shown in Fig. 1. Here the durations of shopping, driving and cooking are uncontrollable (but bounded). This is modeled by using contingent constraints between the start and end events of each action. The fact that the meal should be done within a certain time of the wife's arrival is modeled by a requirement constraint which must be satisfied for the scenario to be correctly executed. The question arising from the scenario is: can we guarantee that the requirement constraint is satisfied regardless of the outcomes of the uncontrollable durations, assuming that these are observable.

In general, STNUs cannot be expected to have static solutions where actions are scheduled at static times in advance. Instead we need dynamic solutions with a mechanism for taking into account the observed times of uncontrollable

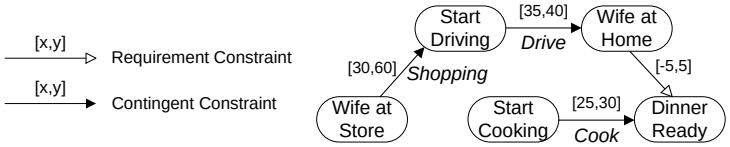

Fig. 1 STNU model of the cooking example.

events (the observed durations of actions). If such a dynamic solution can be found, the STNU is *dynamically controllable* (DC) and the plan it represents can be executed regardless of the outcomes of the contingent constraints.

Example 2 (Continued) The scenario modeled in Fig. 1 does not have a static solution. For every *fixed* time at which cooking could start, there are outcomes for the action durations where the dinner will be ready too early or too late. A dynamic execution strategy exists, however: the man should start cooking 10 time units after observing that the wife starts to drive home. This observation is for instance possible if the wife calls and tells the man that she is about to start driving home. Starting cooking at this dynamically assigned time guarantees that cooking is done within the required time interval, since she will arrive at home 35 to 40 time units after starting to drive and the dinner will be ready within 10+25 to 10+30 time units after she started driving.

Planning with STNUs. Many automated planners begin with an empty plan and then incrementally add one new action at a time using some search mechanism such as forward search or partial-order planning. The initial empty plan is trivially dynamically controllable. If we add an action to a DC plan, the result may or may not be DC. On the other hand, the DC property is monotonic in the sense that if we add an action or a new constraint to a *non-DC* plan, the result is guaranteed *not* to be dynamically controllable. Thus, if the planner generates a non-DC plan at some point during search, extending the plan is pointless. In this situation the search tree can be pruned.

The earlier this opportunity for pruning can be detected, the better. Ideally, the planner should determine after each individual action is added whether the plan remains DC. Dynamic controllability will then be verified a large number of times during the planning process, necessitating a fast verification algorithm. For most of the published algorithms, this would require (re-)testing the entire plan in each step [6,8,9,15]. This takes non-trivial time, and one can benefit greatly from using an *incremental* algorithm instead. The fastest known such algorithm at the moment is the EfficientIDC (EIDC) algorithm [12]. It has a worst-case run-time in $O(n^4)$ and an amortized run-time in $O(n^3)$.

The EIDC algorithm processes nodes one at a time to find all implicit constraints involving them. However, in some situations it will process nodes more than once, leading to inefficiency. In this paper we modify the EIDC algorithm to get the more efficient Efficient^{2}IDC (E2IDC) algorithm. The E2IDC algorithm does not reprocess nodes, leading to a complexity of $O(n^3)$ in the worst case, not amortized.

2 Definitions

We now formally define certain concepts related to STNs and STNUs.

Definition 1 A ***simple temporal network (STN)*** [4] consists of a number of real variables $x_1, \ldots, x_n$ representing events and a set of constraints $T_{ij} = [a_{ij}, b_{ij}], i \neq j$, limiting the distance $a_{ij} \leq x_j - x_i \leq b_{ij}$ between these.

Definition 2 A ***simple temporal network with uncertainty (STNU)*** [17] consists of a number of real variables $x_1, \ldots, x_n$, divided into two disjoint sets of *controlled events* R and *contingent events* C. An STNU also contains a number of *requirement constraints* $R_{ij} = [a_{ij}, b_{ij}]$ limiting the distance $a_{ij} \leq x_j - x_i \leq b_{ij}$, and a number of *contingent constraints* $C_{ij} = [c_{ij}, d_{ij}]$ limiting the distance $c_{ij} \leq x_j - x_i \leq d_{ij}$. For the constraints C_{ij} we require $x_j \in C$ and $0 < c_{ij} < d_{ij} < \infty$.

Definition 3 A ***dynamic execution strategy*** [9] 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.

Definition 4 An STNU is ***dynamically controllable (DC)*** [9] if there exists a dynamic execution strategy for executing it.

Any STN can also be represented as an equivalent *distance graph* [4]. Each constraint [u,v] on an edge $A \rightarrow B$ in an STN is represented as two corresponding edges in its distance graph: $A \xrightarrow{v} B$ and $A \xleftarrow{-u} B$. The weight of an edge $X \rightarrow Y$ then always represents an upper bound on the temporal distance from its source to its target: $time(Y) - time(X) \leq weight(X \rightarrow Y)$. 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 STN [4]. This directly corresponds to the tightest interval constraints $[u', v']$ implicit in the STN. If there is a negative cycle in the distance graph, then no assignment of timepoints to variables satisfies the STN: It is *inconsistent*.

Similarly, an STNU always has an equivalent *extended distance graph* (EDG) [15]. All graphs in this paper, with the exception of Fig. 1, are EDGs of STNUs.

Definition 5 An ***extended distance graph (EDG)*** is a directed multigraph with weighted edges of three kinds: ***requirement***, ***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 described for STNs. Fig. 2 shows an EDG[1] for the cooking example STNU in Fig. 1.

[1] Time is assumed to flow from left to right in all figures.

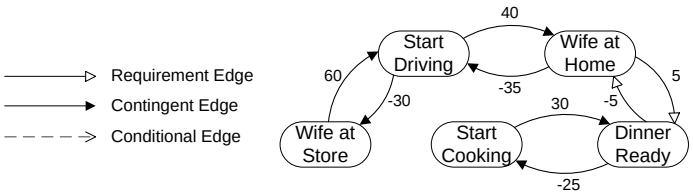

Fig. 2 EDG for the STNU in the cooking example.

A conditional edge [15] is never present in the initial EDG corresponding to an STNU, but can be derived from other constraints through calculations discussed in the following sections.

Definition 6 A ***conditional edge*** [15] $C \rightarrow A$ annotated $< B, -w >$, encodes a conditional constraint: C must occur either after B *or* at least w time units after A. The node B is called the *conditioning node* of the constraint/edge. The edge is *conditioned on* the node B.

We will later see that only conditional edges with negative weights are added to the distance graphs. This is the reason we prefer to annotate these edges with weight $-w$.

A conditional edge means that C must be assigned a time dynamically during execution, when the occurrences of A and B can be observed.

3 DC Verification Techniques

Morris, Muscettola and Vidal [9] were the first to present a way of efficiently (polynomially) verifying if an STNU is dynamically controllable. Their algorithm makes use of STNU-specific tightening rules, also called derivation rules. Each rule can be applied to a triangle of nodes, and if certain conditions are met, new previously implicit constraints are derived and added explicitly to the STNU. It was shown that if these derivation rules are applied to an STNU until quiescence (until no rule application can generate new conclusions), any violation of dynamic controllability can be found through simple *localized* tests. The original algorithm makes intermediate checks while adding constraints to make sure that conditions required for DC are still valid.

The derivation rules of Morris et al. provide a common ancestor theory for most DC verification algorithms, though some exceptions exist (see section 9). The original semantics was later revised [5] and the derivations refined [15]. However, the idea of deriving constraints from triangles of nodes remains.

There are two types of DC verification: full and incremental. Full DC verification is done by an algorithm which verifies DC for a full STNU in one step. Incremental DC verification, in contrast, only verifies if an already known DC STNU remains DC if one constraint is tightened or added. Since incremental algorithms may keep some internal information, it is assumed that the same

Algorithm 1: FastIDC– sound version [10]

```
function FastIDC(EDG G, CCGraph C, edges e_1, ..., e_n)
Q ← sort e_1, ..., e_n by distance to temporal reference
     (order important for efficiency, irrelevant for correctness)
Update CCGraph with negative edges from e_1, ..., e_n
if cycle created in CCGraph then return false
for each modified edge e_i in ordered Q do
    if IS-NON-NEG-LOOP(e_i) then SKIP e_i
    if IS-NEG-LOOP(e_i) then return false
    for each rule from Figure 3 applicable with e_i as focus do
        if applying the rule modified or created an edge z_i in G then
            Update CCGraph
            if cycle created in CCGraph then return false
            if G is squeezed then return false
            if not FastIDC(G, C, z_i) then
            return false
        end
    end
end
return true
```

incremental algorithm is used to process all increments. In this paper we focus only on incremental DC verification.

4 FastIDC

FastIDC is the original incremental DC verification algorithm. Though the first published version was unsound [14], it was later corrected [10]. Algorithm 1 shows the sound version which we for simplicity will refer to only as FastIDC from now on. Understanding how EIDC works requires understanding of FastIDC. We will therefore now describe this algorithm as well as several of its interesting properties.

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 new nodes, resulting in the graph G. FastIDC should then determine whether G is DC. Contingent edges are handled by FastIDC at the time when incident requirement edges are created. Therefore, a contingent edge must be added before any other constraint is added to its target node.

The algorithm works in the EDG of the STNU. First it adds the newly modified or added requirement edges to a queue, Q. The queue 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 will to some extent prevent duplication of effort by the algorithm, but is not essential for correctness or for understanding the derivation process. The algo-

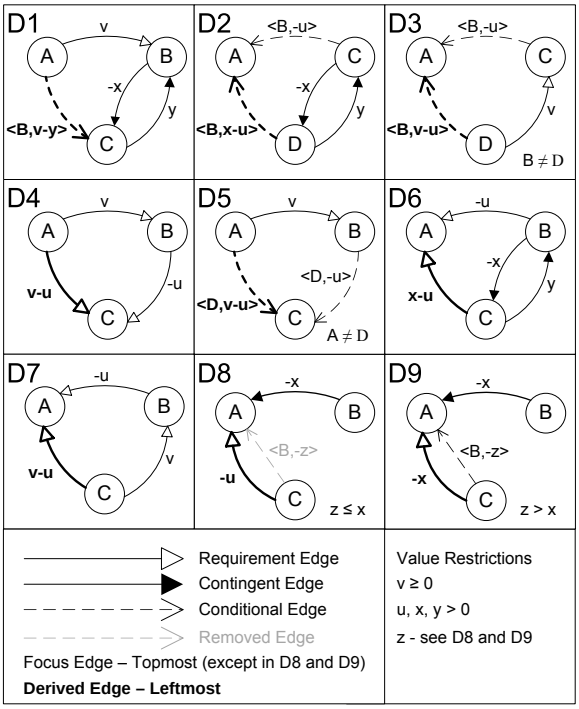

Fig. 3 FastIDC derivation rules D1-D9.

rithm checks that the new edges did not cause a negative cycle, more on this later.

In each iteration an edge e_i is dequeued from Q. A non-negative loop (an edge of weight ≥ 0 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 Fig. 3 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 $\langle B, -u\rangle$. 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, rule D1 will be matched if e_i is a non-negative 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 between the same nodes.

More intuitively, D1 represents the situation where an action is started at the controllable event C and ends at the contingent event B, with an uncontrollable duration in the interval $[x, y]$ where $x > 0$. The focus edge $A \xrightarrow{v} B$

represents the fact that B, the end of the action, must not occur more than v time units after the event A. We see that if B has already occurred, A can safely occur without violating the focus edge constraint. Also, if C has occurred and at least $y - v$ time units have passed, then at most v time units remain until B, so A can safely occur. This latter condition can also be expressed by saying that at most $v - y$ time units remain until C *will* happen (where $v - y$ may be negative). This can be represented explicitly with a conditional constraint AC labeled $\langle B, v - y\rangle$: Before executing A, wait until B or until $-(v - y)$ timepoints after C. This ensures that the fact that A may have to wait for C is available in an edge incident to A, without the need to globally analyze the STNU.

Whenever a new edge is created, FastIDC tests whether a negative cycle is generated. In this case there are events that must occur before themselves. Then the STNU cannot be executed and consequently is not DC. 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 nodes corresponding to the EDG nodes and has an edge between two nodes if and only if there is a *negative* edge between them in the EDG. Note that conditional edges are always accompanied by requirement edges with negative weight (due to the D9 derivation). Therefore, there is never any reason to let these directly affect the CCGraph. Negative contingent edges are however added to the CCGraph. See [10] for further details.

The algorithm then determines if the new edge *squeezes* a contingent constraint. Suppose for example that FastIDC derives a requirement edge $A \xleftarrow{-12} B$, stating that B must occur at least 12 time units after A. Suppose there is also a contingent edge $A \xleftarrow{-10} B$ of weight greater than -12, stating 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 possible outcomes that violate the requirement edge constraint, so the STNU is not DC. The squeeze test is also sometimes referred to as a local consistency check. It involves checking any way that edges between two nodes may cause an inconsistency. Another example is if a positive edge and a conditional edge exist in opposite direction and the sum of the edges' weights is negative. This is also a squeeze. In practice there are many combinations to consider, but they are all carried out in $O(1)$ time.

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 at 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. If FastIDC returns true after processing an EDG, this EDG can be executed directly by a dispatching algorithm [16].

4.1 Properties of FastIDC and its Derivation Rules

We now consider certain important properties of FastIDC. We start with a sketch of the correctness proof.

Theorem 1 *[11] FastIDC correctly verifies whether the STNU remains DC after an incremental change is made.*

Proof (Sketch.) Since FastIDC is not the focus of this paper, we will only provide the intuitions behind the proof here. The full proof is found in [11].

Suppose FastIDC returns false. The rules applied by FastIDC correspond directly to the sound rules of the original full DC verification algorithm [9], so all new constraints that are derived are valid consequences of the information that is already in the STNU. Since FastIDC returned false, applying these sound rules must have resulted in a squeeze or a negative cycle. Then the STNU cannot be DC, and FastIDC returned the correct answer.

Suppose FastIDC returns true. Before the edges $e_1, \ldots, e_n$ were added or tightened, the STNU was dynamically controllable. For each edge e_i in this set, FastIDC applies all possible derivation rules with e_i as a focus, thereby deriving all possible *direct* consequences of the addition or tightening. When this results in new additions or tightenings of edges z_i, the algorithm recursively handles these *indirect* consequences in the same way. This is sufficient to derive *all* consequences that can be derived using the specified derivation rules.

It can be shown that if all consequences of a set of modifications are derived and added to an STNU, and if this does not lead to a negative cycle or a squeeze, then there exists a dynamic execution strategy for the STNU. Abstractly, the reason for this is that (a) if the STNU had been inconsistent in the STN sense, the derivations would have resulted in a negative cycle, and (b) if uncertain durations could have had outcomes that led to violations of requirement constraints, then the derivations would have resulted in a squeeze. Since FastIDC returned true, this did not happen. Then the STNU must be DC, and FastIDC returned the correct answer. □

Complexity. The efficiency of FastIDC depends on the order in which edges are selected for processing. Intuitively, the recursive derivation procedure FastIDC uses leads to derivation chains that can be circular, so that tightenings are often applied repeatedly to the same subset of edges. These edges will eventually converge to their final weights, but some edge orderings will result in faster convergence than others. The effect of order on run-time is examined at length in [12] where it is shown that the best possible efficiency attainable by FastIDC comes from modifying the algorithm to keep a global queue and processing edges from the end of the STNU towards the start. However, even with this modified approach, FastIDC has a worst case complexity of $\Theta(n^4)$ per tightened edge [12].

Edge Interactions. The derivation rules prevent constraints from being violated by placing new constraints on *earlier* nodes. These new constraints must

be satisfied if the STNU is dynamically controllable. A side effect of this is the following result:

Lemma 1 (Plus-Minus) *Except for rules D8 and D9, derivation of new edges requires the interaction of a non-negative and a negative edge. The derived edge has either the same source (D1, D4, D5) or the same target (D2, D3, D6, D7) as the focus edge used in its derivation. If the source stays the same, the target coincides with that of the negative edge. If the target stays the same, the source coincides with that of the non-negative edge.*

Proof By inspecting the derivation rules D1-D7, it is clear that the existence of a non-negative edge followed by a negative edge is required in all cases. It is also seen that the sources and targets behaves as stated in the lemma. □

Note that calling the lemma Plus-Minus is not entirely accurate since it leaves out the fact that first edge may have zero weight.

Note also that it is not stated in the lemma which weights are actually used for deriving the new edge. Rule D1 has plus-minus interaction, but the value used to find the weight of the derived edge comes from the positive contingent edge.

The important property captured by the lemma is that the negative edge must exist, which gives a structure to the derivations: in D1, C must occur before B, which leads derived edges toward the start of the EDG.

Effects of the Plus-Minus Lemma are studied in detail in a previous paper [11], but the lemma is not mentioned there directly. Instead positive-negative or plus-minus interaction were mentioned as an observation in the correctness proof for EIDC [12].

5 The EfficientIDC Algorithm

FastIDC may derive edges between the same nodes several times, which is problematic for its performance. Though there are cases where this can be prevented to a certain degree [12], it remains a problem to efficiently handle derivations in regions containing many unordered nodes, i.e. nodes that are mostly connected by non-negative requirement edges.

To overcome these problems a new algorithm was proposed [12]: The Efficient Incremental Dynamic Controllability checking algorithm (Algorithm 2, EfficientIDC or EIDC for short). EIDC will now be described in some detail, as it is the basis for the improved algorithm presented in this paper.

EIDC uses focus *nodes* instead of focus edges to gain efficiency. It is based on the same derivations as FastIDC (Fig. 3) but applies them differently. When EIDC updates an edge in the EDG, the target of the edge, and the source in some cases, are added to a list of focus nodes 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. This guarantees that no tightenings are missed [12].

The focus node processing is made possible by the Plus-Minus Lemma. As an example, suppose we have a negative edge $A \rightarrow B$. If we derive a new edge along this negative edge, which means that there was a non-negative edge targeting A by the lemma, further derivations based on this derived edge cannot come back to target A (unless non-DC). This follows since, by the lemma, the target of derived edges follows negative edges and hence coming a full circle back to the starting position requires the existence of a negative cycle. Therefore, if nodes are processed in the optimal order there will be no later stage of the algorithm where a previously derived edge is replaced by a tighter edge. The behavior of derivation chains (i.e. derivations caused by derivations), including a detailed proof that derivations cannot cause cycles has been previously published [11].

EIDC has a worst case run-time for one call in $O(n^4)$. However, this worst case cannot occur frequently. Therefore the complexity can be amortized to $O(n^3)$ per increment. This is a significant improvement over FastIDC which is either exponential or $\Omega(n^4)$ depending on the algorithm realization [12].

The use of a focus node allows EIDC 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 [12]. The key to understanding this is that derivation rules essentially calculate shortest distances. For example, rule D4 states that if we have tightened edge $A \rightarrow B$ and there is an edge $C \leftarrow B$, an edge $A \rightarrow C$ may have to be tightened to indicate the length of the shortest path between A and C. Dijkstra's algorithm cannot be applied indiscriminately, since there are complex interactions between the different kinds of edges, but can still be applied in certain important cases.

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

As in 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. EIDC accepts one tightened or added edge, e, in each increment. If several edges need to be added, EIDC must be called for each change.

The EfficientIDC algorithm. The EIDC algorithm is shown in Algorithm 2. 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. If this causes a negative cycle in the CCGraph, G is not DC. Otherwise, Source(e) is also added for processing. This is because in order to find all incoming edges to Target(e) all nodes after Target(e) must have been processed before Target(e) itself.

Iteration. As long as there are nodes in *todo*, a node to process, *current*, is selected and removed. The chosen node must not have incoming edges in the

Algorithm 2: The EfficientIDC Algorithm

```
function EfficientIDC(EDG G, DDG D, CCGraph C, Requirement 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, current)
    ProcessPosReq(G, current)
    for each edge e added or modified in 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
```

CCGraph from any node which is currently in the *todo* set. This requirement forces the algorithm to process temporally later nodes before temporally earlier ones, which means that when the earlier nodes are processed, there will be no new edges appearing behind them. Therefore, assuming optimal choices, the algorithm can finalize nodes as they are processed iteratively. How non-optimal choices affect the complexity will be discussed later.
As long as *todo* is not empty, there is always a *todo* node satisfying this criterion. If not, there would have been a cycle in the CCGraph and consequently a negative cycle in the EDG, a fact which would have been detected.

When *current* is assigned, assuming optimal processing order, we are sure that we have found all externally incoming edges to *current* and it is time to process it using the three helper functions shown in Algorithms 3 to 5. This can derive even more incoming edges and also add some edges targeting earlier nodes. Processing *current* thereby determines which earlier nodes will need processing due to their newly derived incoming edges. These are added to *todo*.

Incoming conditional edges are processed similarly to FastIDC focus edges using ProcessCond. This is equivalent to applying rules D2, D3, D8 and D9, but is done for a larger part of the graph in a single step compared to FastIDC. There are only $O(n)$ contingent constraints in an EDG and hence only $O(n)$ conditioning nodes (nodes that are the target of a contingent constraint). All

Algorithm 3: Process Conditional Edges

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

times in conditional constraints/edges are measured towards the source of the contingent constraint. Therefore, all conditional constraints conditioned on the same node have the same target.

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

For each conditioning 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 source shortest path tree rooted in *current*. Derivations over non-negative requirement edges traverse the edges in reverse order, and so the DDG contains these edges in reverse order. Derivations over contingent edges follows the negative contingent edge, but the distance used in the derivation is the positive weight of this, so this is also contained in the DDG. The section of the graph which can be traversed contains only non-negative weight edges and so Dijkstra's algorithm can be used to find the shortest paths. The only remaining issue is that the edges connecting the source of the tree we want to build are negative and in reverse order. Since only one of these edges will be used by each path, there is no risk of negative cycles so they could be used directly. However, when EIDC reverses the edges it also adds a positive weight to them to make all edges used by the Dijkstra calculation non-negative. The added weight, *minw*, is the absolute value of the most negative edge weight of the incoming conditional edges. This value also serves as a cut-off for stopping the Dijkstra calculation. Once the distance is longer than *minw* the derived result will be a positive edge which cannot further react to cause more derivations. Running Dijkstra calculations will in a single call derive a final set of

shortest distances that FastIDC might have had to perform a large number of iterations to converge towards. An example in the next section shows how this is carried out. We will see a detailed implementation of the LimitedDijkstra function in section 8.

After this the algorithm 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. It then applies the "special" derivation rules D8 and D9, which convert conditional edges to requirement edges. Note that 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*, and must therefore 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 thus 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.

Algorithm 4: Process Negative Requirement Edges

```
function ProcessNegReq(EDG G, DDG D, Node current)
edges ← IncomingNegReq(current, G)
minw ← |min{weight(e) : e ∈ edges)}|
add minw to the weight of all e ∈ edges
for e ∈ edges do
    add e to D with reversed direction
end
LimitedDijkstra(current, D, minw)
for all nodes n reached by LimitedDijkstra do
    e ← req. edge (n → current) of weight Dist (n) - minw
    if e is a tightening then add e to G
end
Revert all changes to D
return
```

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

Incoming positive requirement edges are processed using ProcessPosReq, which applies rules D1, D4 and D5. These are the rules that may advance derivation towards earlier nodes. By deriving a new edge targeting an earlier node, the node is put in *todo* by the main algorithm.

After processing incoming edges. These are the only possible types of focus edge in FastIDC derivations (Fig. 3). Therefore all focus edges that could possibly have given rise to the *current* focus node have now been processed.

Algorithm 5: Process Positive Requirement Edges

```
function ProcessPosReq(EDG G, Node current)
for each e ∈ IncomingPosReq(current, G) do
    apply derivation rule D1, D4 and D5 with e as focus edge
    for each derived edge f do
        if f is conditional edge then
            apply derivations D8-D9 with f as focus edge
        end
        if derived edge is a tightening then
            add it to G
        end
    end
end
return
```

EIDC then checks all edges that were derived by the helper functions. 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. It 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 will be added to *todo*.

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

Updating the CCGraph. A novel feature of EIDC as compared to FastIDC is that the CCGraph now contains the *transitive closure* of all edges added to it. This prevents reprocessing when new orders are found through ProcessPosReq. How the transitive closure is derived and the gains from using it will be discussed later.

Updating the DDG. The DDG contains weights and directions of edges that FastIDC derivations use to derive new edges, and is needed to process edges effectively. Edges in the DDG have no type, only weights that are always positive. 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 to the DDG have been omitted. Updating the DDG is straight forward and 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 (with the absolute value of their weight as new weight). In case a positive requirement edge disappears from the EDG it is removed from the DDG.

Complexity. The complexity of EIDC depends on how many new orderings are discovered while processing nodes [12]. If no new orderings between nodes

are discovered the algorithm runs in $O(n^3)$ since the total processing of conditional edges takes $O(n^3)$ and the rest of the derivations takes $O(n^2)$ per node for a total of $O(n^3)$. New orderings involving *current* that are discovered when processing *current* will cause *current* to be reprocessed. However, there is no need to do this if the discovered "later node" was already processed in the right order, i.e. before *current*. This will be handled in the new version of the algorithm, to be presented later.

Each new ordering requires a negative requirement edge between the nodes involved. This limits the number of such reprocessings to n^2 times in total and bounds it by n per node. This gives an upper bound of $O(n^4)$ for all possible reprocessings of requirement edges. The bound for reprocessing conditional edges also becomes $O(n^4)$ since each such edge will only be reprocessed when its target node is reprocessed and this happens at most n times per conditioning node. In the next section we give an example of how EIDC processes an STNU. We then follow this up with an example showing that reprocessing cannot be avoided by EIDC, a fact which motivates the presentation of Efficient^{2}IDC in section 8.

Correctness. We end this section with a short sketch of EIDC correctness, which is a building block for Efficient^{2}IDC correctness. Correctness for EIDC builds on the fact that it generates the same EDG as FastIDC.

Theorem 2 *[12] EIDC correctly verifies whether the STNU retains DC after an incremental change is made.*

Proof (Sketch) Soundness follows since the derivations performed by EIDC are either through direct use of FastIDC derivation rules or through the use of Dijkstra's algorithm in a way that corresponds directly to repeated application of derivation rules. Since these rules are sound, EIDC is sound in terms of edge generation.

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, this will also become the case 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 dynamic controllability in the same way they also agree on which STNUs are DC and which are not. □

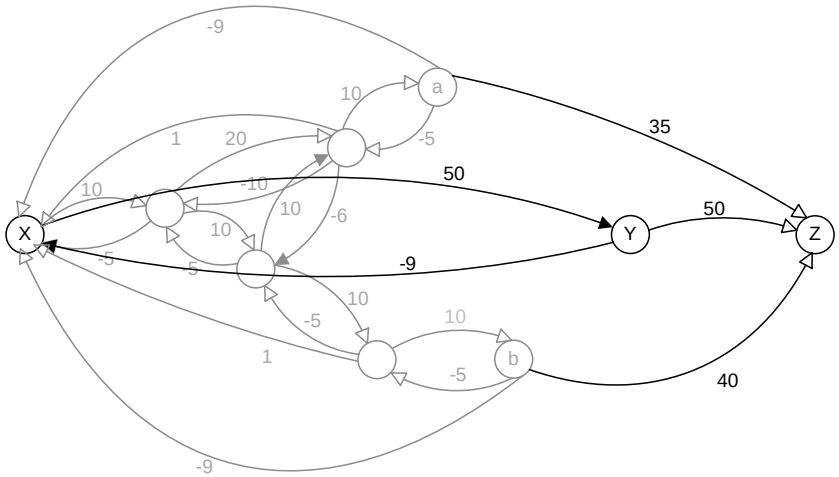

Fig. 4 Initial EDG.

6 EfficientIDC Processing 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.

Fig. 4 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.

In the example we add a new requirement edge[2] $Y \xleftarrow{-10} Z$ as shown in the rightmost part of Fig. 5. When we call EIDC for this edge, both Y and Z will be added to *todo*. Z must be processed first because of the ordering between Z and Y. Since Z has no incoming conditional or negative requirement edges only ProcessPosReq will be applied. This results in the bold requirement edges $a \xrightarrow{25} Y$ and $b \xrightarrow{30} Y$. The node Y is then selected as *current* in the next iteration. Even though Y has an incoming negative edge, no new derivations are done by ProcessNegReq. However, Y also has two incoming positive requirement edges that are processed (using D1) to generate the conditional edges $X \xleftarrow{\langle Y,-25\rangle} a$ and $X \xleftarrow{\langle Y,-20\rangle} b$. Two negative requirement edges, $X \xleftarrow{-9} a$ and $X \xleftarrow{-9} b$, are also derived alongside the conditional edges due to D9 but these are not stronger than the already existing identical edges. Since there were already edges $X \leftarrow a$ and $X \leftarrow b$ in the CCGraph, a and b are not added to *todo*. However, X is added as the target of a newly derived edge is always added to *todo*. Since the derived edges are not incoming to Y they require no

[2] When possible we will use the arrow direction from the figures in the text.

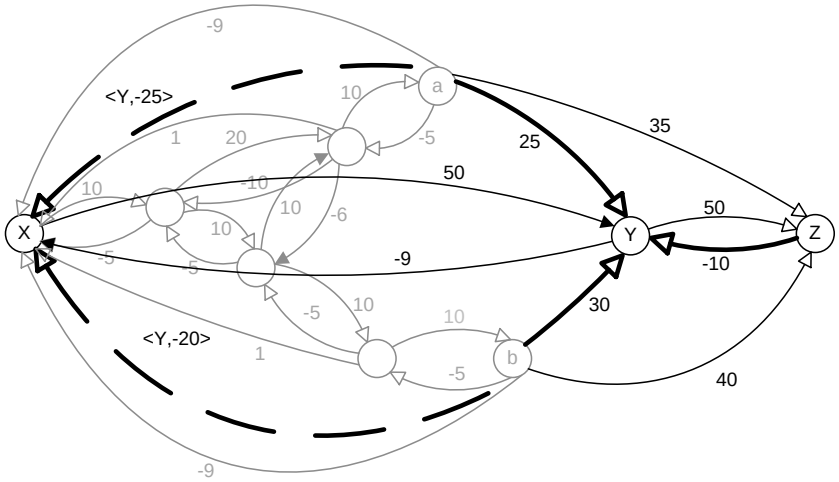

Fig. 5 Derivation of the smaller scenario.

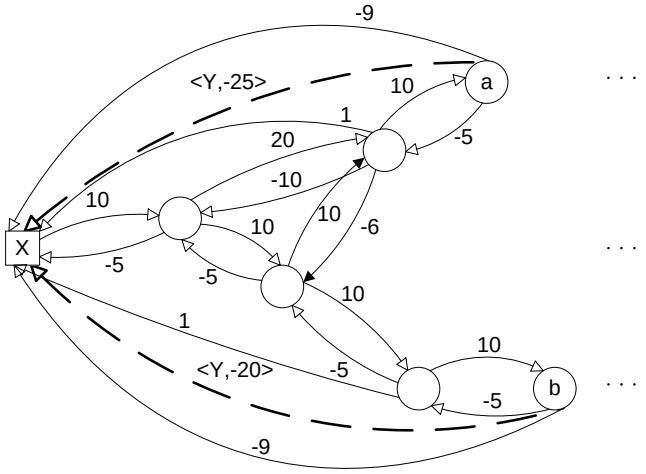

Fig. 6 Example scenario for conditional edges.

further processing at the moment. This leaves only X in the *todo* set for the next iteration.

In the next iteration, X is selected as *current*. No more edges will be derived in the rightmost black part of the example EDG, so we focus on the previously gray part of the EDG shown in Fig. 6. We see that X has two incoming conditional edges with the same conditioning node Y. These edges are processed together, resulting in a *minw* value of 25. After adding edges corresponding to the reversed conditional edges, each with a weight increase of *minw*, we get the DDG that is used for Dijkstra calculations when

processing X. The DDG is shown in Fig. 7. Recall that in the DDG all positive edges are present with reversed direction and all negative contingent edges are present with positive weight. Note that the weight 1 edges from X are left out of the DDG in Fig. 7. These are present in the DDG but cannot be used when X is *current* since using them would require that the source and target of the conditional edge used for derivation was the same. This is a degenerate case which cannot occur in the EDG. Such an edge would either be removed before addition or responsible for non-DC of the STNU. In Fig. 7 we have labeled each node with its shortest distance from X in the DDG.

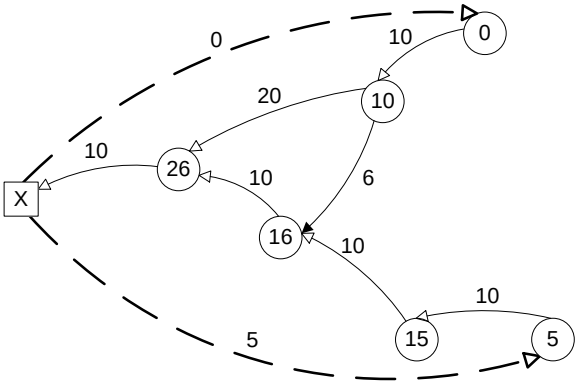

Fig. 7 Dijkstra Distance Graph of the small scenario.

Processing $current = X$ gives rise to the bold edges in Fig. 8. We consider how the -9 edge is created. First the distance from X to the source node of the -9

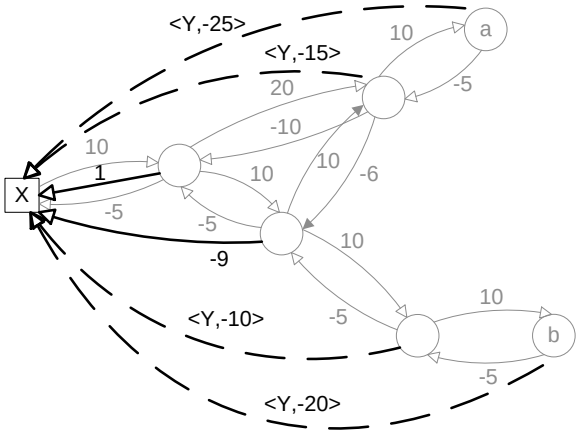

Fig. 8 Result of processing $current = X$.

edge is calculated by Dijkstra's algorithm. This is 16 (see Fig. 7). Subtraction of 25 gives a conditional edge with weight -9. However, since the lower bound of the contingent constraint involving X is 9, 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, starting with the conditional $X \xleftarrow{\langle Y,-25\rangle} a$ edge as focus.

The example shows how EIDC adds *minw* to the negative edges from the source to get non-negative edges for Dijkstra's algorithm to work with.
Finally, all new derived edges need to be checked so they do not squeeze existing edges, and negative edges should be added to the cycle checking graph when needed.

7 Reprocessing by EfficientIDC

The following example shows a situation in which EIDC could reprocess a node. The STNU involved, shown in Fig. 9, contains the following two components: A split which causes derivations to take two alternate paths toward X, and a region of non-negative edges where the nodes can be processed in a non-optimal way.

Suppose that an edge $a \xrightarrow{17} g$ is added as an incremental change and EIDC is called. This first adds g to *todo*. When g is chosen for processing, the new incoming edge is combined with the two outgoing edges, so the new edges $X \xleftarrow{-20} a$ and $a \xrightarrow{7} f$ in Fig. 10 are derived through the ProcessPosReq function. A consequence is that a, f and X are added to *todo*.

EIDC could then choose to process a first, which leads to addition of the edges $X \xleftarrow{-17} b$ and $X \xleftarrow{-10} c$. Since these two edges are negative and not in the CCGraph both b and c are added to the *todo* set. At this point *todo* =

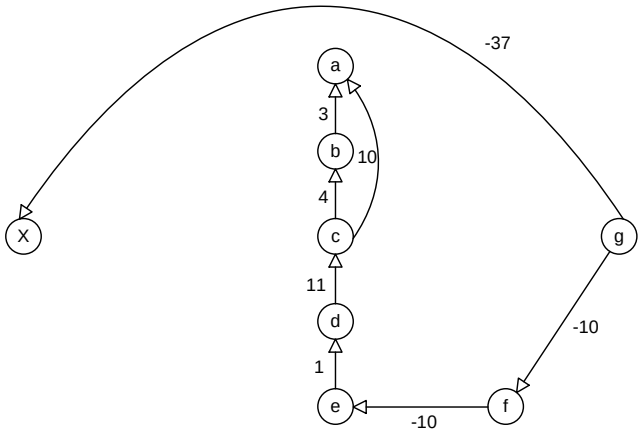

Fig. 9 Start situation of the EIDC reprocessing example.

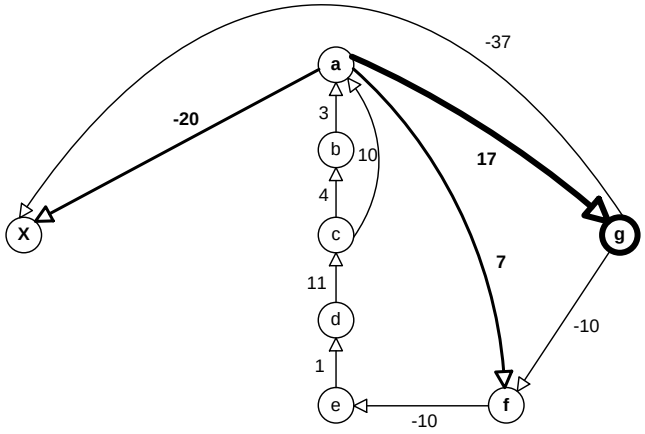

Fig. 10 Processing the $a \rightarrow g$ edge.

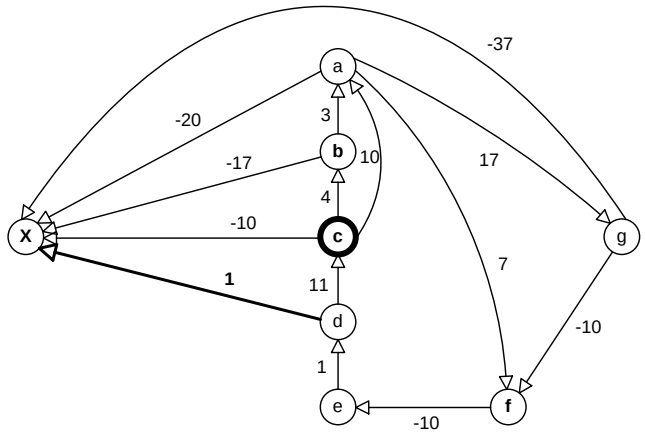

Fig. 11 After just processing a and c.

$\{b, c, f, X\}$. EIDC could then choose to process c which adds only the $X \xleftarrow{1} d$ edge. Note that d is not added to *todo* since sources of non-negative edges are not added to *todo* by EIDC. The situation at this point is shown in Fig. 11.

If EIDC processes b as its next *current* it will derive the edge $X \xleftarrow{-13} c$. This tighter edge will replace the existing edge $X \xleftarrow{-10} c$, but since the corresponding *ordering* between X and c was already known, the source c will not be added to *todo* by EIDC. It only attempts to add the target, X, which in this case is already present.

At this point the cause for reprocessing is passed, namely that c was processed before b. This leaves the only way of finding the order between X and e to be by a ProcessNegReq derivation with X as *current*. The final situation when this happens is shown in Fig. 12.

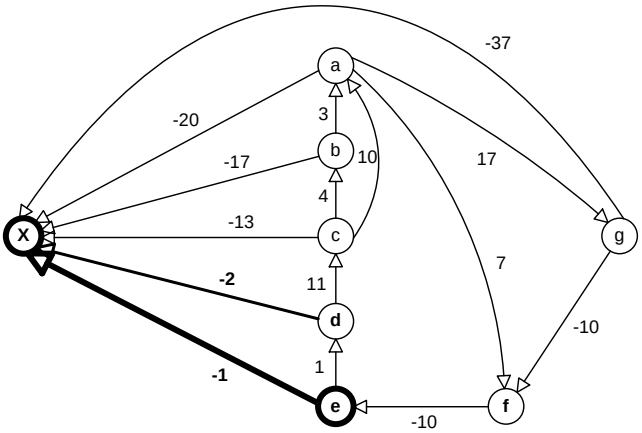

Fig. 12 The final situation when the order between X and e is discovered.

In the example, the ordering between X and e is only found when the negative requirement edge $X \xleftarrow{-20} a$ has reacted along the shortest DDG path from a to e. For EIDC to avoid reprocessing this has to happen before X is processed at which point it is too late. Reprocessing of X is required since EIDC adds both source and target to *todo* when a new ordering is found. It is possible that EIDC chooses the nodes in an optimal order, but it cannot be guaranteed or even expected.

We can use the example as an inspiration for a more efficient algorithm. If we look at the distances calculated by Dijkstra's algorithm when processing X we can see that distances from X to nodes visited by Dijkstra prior to reaching e, are not affected later by the fact that e is processed. By this we mean that processing e does not add tighter edges to the path between a and e. This is not a coincidence pertaining only to this example. In the next chapter we prove that this is universally true and it becomes the basis for how the Efficient^{2}IDC algorithm avoids reprocessing nodes.

From the example we can also see the two possible ways of discovering a new ordering:

1. By applying ProcessPosReq which discovers the order between a and X in the first step.
2. By applying ProcessCond or ProcessNegReq which discovers the order between X and e.

We will refer to these sources of discovery as type 1 and 2 discoveries. We will also refer to derivations as type 1 and type 2. In the next section we will show a way of dealing with these new orderings in a way which allows an $O(n^3)$ worst case complexity.

We end this section by remarking that if EIDC always added the source of negative/conditional edges to the todo-set instead of doing this only the first time an order is discovered, all orderings would be discovered by Process-

PosReq. In fact, EIDC would then become more similar to FastIDC. In case a region of non-negative edges was encountered, like in the example, the shortest paths would then be derived from the source side since each time a possible tightening is found the source would be added until the shortest paths in the region were found, at which time there would be nothing to do for ProcessCond or ProcessNegReq when X is processed. However, EIDC would then have the same problems as FastIDC [12] where edges would be overwritten iteratively as the weights approached the tightest values, giving the algorithm an $O(n^4)$ run-time complexity.

If, on the other hand, EIDC never added the source, a new ordering of type 2 would not cause a reaction. This could lead EIDC to do much work stemming from the target before the ordering was discovered by a type 1 discovery later. At this point, all the previous work would have to be redone.

While none of the discussed alterations render EIDC incorrect they both impact performance. It seems that processing the source exactly once is a good idea for discovering as many type 1 new orderings as possible without the algorithm becoming $O(n^4)$. In the next section we see how we can remove the need for reprocessing altogether.

8 The Efficient^{2}IDC Algorithm

In this section we present the Efficient^{2}IDC (E2IDC) algorithm. It is an improved version of EIDC which is $O(n^3)$ non-amortized, even in the worst case. The intuition behind the improvement is that some of the derived edges will not be affected by the discovered ordering. Therefore, a full reprocessing of the target node is not needed. Instead the algorithm can pause the processing of the target node and come back to finish it later. The resulting EDG of E2IDC is identical to that of EIDC, so correctness follows directly. We use the example in the previous section to explain the idea. Throughout we will also refer to type 1 and 2 derivations/discoveries. The externally added edge of an increment does not fall into either of these categories since it is not derived. However, it can act as both, for instance by adding only the source to *todo* or both source and target. In the remainder of this section we will treat the external edge as derived by both a type 1 and a type 2 derivation to cover all cases. We start by a definition applicable to both algorithms.

Definition 7 If a node is processed by EIDC or E2IDC, to be presented later, during which no type 2 discovery is made, the node is said to be *completely processed*.

We would like to clarify the usage of *new* and *tightening* in the coming discussions. By regarding an edge not present in the EDG as present with infinite positive weight we can label a "new" derived edge as a tightening of an existing edge. This lets us simplify the presentation, but we need to take care when the derived edge weight between two nodes becomes negative for the first time. This is identified as a new ordering as well as a tightening. Note that from

an ordering perspective it does not matter which type of edge that is derived between two nodes. An ordering follows from a negative requirement edge, a negative contingent edge or a conditional edge with negative weight.

We are now ready to continue with several lemmas that apply to EIDC and can be transferred to the new E2IDC algorithm which will be presented after.

Lemma 2 (Requirement Lemma) *Suppose that the node n was completely processed at some point, in this increment or previously, and that n is now processed again. For this processing to result in the tightening of an incoming edge to n the following is required: there must be a tightened incoming edge to n compared to when it was last completely processed.*

Proof Note that deriving an incoming edge towards n, when processing n, requires a type 2 derivation since type 1 derivations that are done when processing n targets other nodes. We will therefore show that any type 2 derivation requires a tightened incoming edge compared to when the node was last completely processed.

Suppose towards contradiction that no incoming edge was tightened and it is still possible to make a type 2 derivation. If no incoming edge was tightened, then all incoming edges have the same weight as the last time n was processed. Derivations of type 2 takes place when the source of a negative or conditional edge is derived along a non-negative distance in the DDG. For the type 2 derivation to occur there must be an involved negative or conditional edge $e_1 = n \xleftarrow{-a} s$. Furthermore, there must also be a non-negative distance from the source of this edge in the DDG. Therefore, there must either be a contingent edge $e_2 = t \xleftarrow{-b} s$ or a non-negative requirement edge $e_3 = t \xrightarrow{c} s$.

Derivation of a tighter edge when processing n requires that one of the edges involved in the derivation is tighter than it was the last time n was processed.

Since the weight $-a$ has not changed by the assumption, the other edge must be tightened compared to before. A contingent edge cannot be tightened, that would mean it is squeezed and the STNU become non-DC. Therefore, the situation must be that $e_3 = t \xrightarrow{c} s$ is present and the weight c is less than it was when n was processed previously.

At some point after n was completely processed previously, s received a tighter incoming edge. The node s was therefore put in *todo* and processed. When processing s, ProcessPosReq would have derived the result of combining e_3 with the also present e_1 edge. This would then have given a tighter incoming edge to n which contradicts the original assumption. Therefore we conclude that any type 2 derivation when processing n requires the prior tightening of an incoming edge to n. □

We now specify the origin of the incoming edge in a corollary:

Corollary 1 *The required incoming edge in the Requirement Lemma must be the result of a type 1 derivation.*

Proof Type 2 derivation only derives edges that targets the node being processed. Therefore, n cannot receive a tighter incoming edge from a type 2 derivation when another node is processed. The same also holds regarding type 2 derivations when processing n, since any type 2 derivation requires the presence of an already tightened incoming edge by the lemma. Therefore, the first derived tighter edge which targets n must be the result of a type 1 derivation. □

We follow this with putting the focus on a property of the EIDC algorithm.

Property 1 (Blocking Property) Suppose that n is ordered before m, i.e. there is a negative edge $n \leftarrow m$. If m is put in *todo*, from this point on, n cannot be processed before m is completely processed.

Proof Because of the ordering, m has to be removed from *todo* before n can be considered. If time m is processed but does not get completely processed, a new order of type 2 is found. This causes m to be put back into *todo*, and it continues to block n.

If processing m makes it completely processed, m will not enter *todo* again before giving EIDC the possibility of processing n. □

The Blocking Property means that m temporarily blocks n. It follows from the property that any node which blocks m, due to transitivity in the CCGraph, also blocks n.

We will now see that the definition of completely processed meets the expectations.

Lemma 3 (Finished Lemma) *If a node is chosen from todo and becomes completely processed, further processing by EIDC in this increment cannot cause derivation of tighter incoming edges to it.*

Note that the lemma does not state that n cannot be added to the *todo* set multiple times. It states that if it should be added after the situation described, no tighter derivations will be made.

Proof We assume that the node, n, is chosen from *todo* and becomes completely processed. We now proceed to show that no tighter incoming edge to n can be found in this increment.

There are two situations in which an incoming edge to n could be derived. Either when processing n, through a type 2 derivation, or through another node ordered after n where a type 1 derivation causes the incoming edge to be derived. The Requirement Lemma states that for n to derive an incoming edge through a type 2 derivation, a type 1 derived edge towards n must first be derived. Therefore, in both cases the possibility for a tightened edge towards n relies on finding a type 1 derived edge. We will continue to show that no such edge can be derived.

A type 1 derived edge would have to be derived when processing a node m that has a negative edge $n \leftarrow m$ towards n. We can assume without loss of

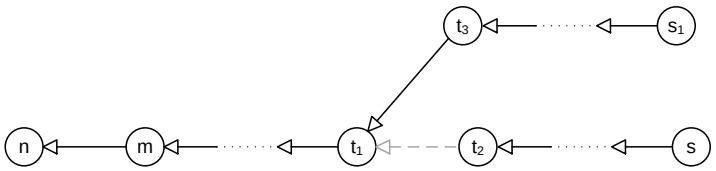

Fig. 13 Reasoning about processing chains. The edges correspond to negative weight edges in the EDG.

generality that m is the first node, after n was completely processed, that when processed causes the derivation of a type 1 derived edge towards n. If there is an edge derived, there must be a first such edge. Since m is the first node which causes the derivation of an edge towards n the $n \leftarrow m$ edge must have been present when n was processed. Because, if this edge was derived when processing another node, x, after n was completely processed, the $n \leftarrow m$ edge would be derived before m was processed, contradicting the assumption on m.

Since the $n \leftarrow m$ edge is present when n becomes completely processed, we know that at this time m cannot be in the *todo* set. But in order to be processed, m must have entered the *todo* set. Therefore, another node, s, must be in the *todo* set when n is processed, and processing s at a later time will start a processing chain that ends up with m being put in the *todo* set.

We now study this processing chain in detail. Fig. 13 can be used to follow the reasoning. We first remind the reader that the target of the derived edges follows negative edges. Therefore, in order for a type 1 derivation to eventually target n, there must at some point be a negative path from s to n. However, there cannot be a negative path present all the way from s to n before n is processed as the transitive closure handling would ensure that the $n \leftarrow s$ edge would be present in the CCGraph, blocking n from being processed until the sought type 1 edge was already in place. This contradicts that the edge is derived after n is processed. Therefore, at least one of the negative edges required along the chain must be derived after n has been completely processed.

We now assume the "missing" edge closest to n is $t_1 \leftarrow t_2$. This means that none of the nodes along the negative path between t_1 and n could have been in *todo* when n was processed. Due to the Requirement Lemma, if the $t_1 \leftarrow t_2$ edge is caused by a type 2 derivation, a type 1 derivation is required prior. Thus, regardless of which type of derivation is responsible for the edge, there must be a node t_3 ordered after t_1 by a negative edge, which facilitates the derivation of the missing edge. For this edge to be derived, t_3 must be put in *todo* after n is completely processed. But there was a chain of negative edges from t_1 to n when n was chosen to be processed. This means that $n \leftarrow t_1$ existed in the CCGraph and so also for $n \leftarrow t_3$. Therefore, as for m, there must again be a node s_1 (which might be equal to s) which is responsible for starting the chain that ends with t_3 being put in *todo* so that it may later cause derivation of the $t_1 \leftarrow t_2$ edge. Now, since t_3 has a negative path to n this cannot be the case for s_1 by the same reasoning as for s. Therefore, another

edge must be missing at the moment between t_3 and s_1. If we continue the same reasoning as before we see that each missing edge requires a new t-node which requires another missing edge and so on. This leads to a growing chain of different t nodes until one of the t nodes must equal one of the s nodes. At this final point we see that there is a chain of negative edges from one of the s nodes in *todo* to n which contradicts the possibility of a missing edge and ultimately a derived edge towards n. □

The lemma will be used by E2IDC to improve the algorithm's efficiency. We are now ready to present this algorithm in listings 6-8.

The algorithm is a modified version of EIDC. We now go through the main points of modification.

Recursion. A consequence of the Finished Lemma is that if E2IDC process a node *current* and do not find any new type 2 ordering it is safe to move on and process nodes before *current*. If a new type 2 order is found, the source node must be processed before E2IDC can continue to process *current*. This is done recursively, which leads to a division of the main algorithm into the main iteration loop and a ProcessNode function. This can then be called recursively from the LimitedDijkstra function when new orders are detected.

The *finished* set. The Finished Lemma directly tells us that any node that is completely processed does not need to be processed again. A call to the function ProcessNode has this effect for the processed node. To increase the efficiency of the E2IDC algorithm, completely processed nodes are kept in a *finished* set. If a node in the *finished* set is encountered in a type 2 derivation, we now know that reprocessing it will not give it any tighter incoming edges, and so this encounter should not cause any reprocessing.

The *processing* set. This set keeps track of nodes that are being processed. If more than one node is in the set, recursive processing is ongoing. The set has two uses. First, it prevents recursively processed nodes from adding nodes earlier in the recursion chain to the *todo* set. These nodes are already being handled. Second, if a negative cycle is derived via recursive calls to ProcessNode and LimitedDijkstra this recursion may loop unless the cycle is detected. Detection in this case happens when LimitedDijkstra finds a new ordering for which the source node is already in the *processing* set.

The TCGraph. To ensure the correctness of FastIDC, negative cycles had to be detected. The CCGraph was introduced for this purpose [10], and was updated using a fast but complex incremental topological ordering algorithm [1]. EIDC additionally needs to keep track of the transitive closure of all negative edges. The transitive closure can of course directly be used to find cycles of negative edges, which makes the CCGraph redundant. In E2IDC we therefore change from the CCGraph as cycle detector to the TCGraph, containing the transitive closure of all negative edges.

From the Finished Lemma we know that no edges that targets a node will be generated after it is completely processed. Therefore, the transitive closure can be updated to include all the effects of processing a node at the end of

Algorithm 6: The Efficient^{2}IDC Algorithm

```
function Efficient²IDC(EDG G, DDG D, TCGraph C, Requirement Edge e)
    finished ← {}
    processing ← {}
    todo ← {Target(e)}
    /* With the exception of e, all in-parameters and the sets finished,
       todo and processing are considered globally visible. G, D and C
       are modified by the algorithm.                                    */
    if e is negative and e ∉ C then
        add e to C
        if negative cycle detected then return false
    end
    while todo ≠ ∅ do
        current ← pop some n from todo where
            ∀e ∈ Incoming(C, n) : Source(e) ∉ todo
        if ProcessNode(G, D, C, current) = false then return false
    end
    return true

function ProcessNode(EDG G, DDG D, TCGraph C, Node current)
    processing ← processing ∪{current}
    ProcessCond(G, D, C, current)       // Edges target current. Applies D2,D3
    ProcessNegReq(G, D, C, current)     // Edges target current. Applies D6,D7
    ProcessPosReq(G, C, current)           // Other targets. Applies D1,D4,D5
    for each edge e added to G while processing current do
        if e is a non-negative requirement edge then add e to D
        if Target(e) ≠ current and Target(e) ∉ processing then
            todo ← todo ∪{Target(e)}              // Target needs processing
        end
        if e is a negative requirement edge and e ∉ C then
            add e to C
            remove e from D if present
            if negative cycle detected then return false
            if Source(e) ∉ finished then     // Process unprocessed nodes only
                todo ← todo ∪{Source(e)}
            end
        end
    end
    if G is squeezed then return false
    finished ← finished ∪ {current}
    processing ← processing − {current}
    UpdateTCGraph ()
    return true
```

ProcessNode when it is completely processed. Each call to ProcessNode use up to $O(n^2)$ time. This means that E2IDC may use any algorithm within this complexity class for generating the transitive closure.

It turns out that the naive algorithm is enough to meet our requirement: First, all negative requirement edges that was derived in this iteration are added to the TCGraph. Then find all edges that targets *current* in the TC-Graph and the predecessors of their sources. These are then connected in

Algorithm 7: The Edge Derivation Functions

```
function ProcessCond(EDG G, DDG D, TCGraph C, Node current)
                                                  // Indirectly applies D2 and D3
    allCond ← IncomingCond(current, G)            // conditional edges into current
    condNodes ← {n ∈ G | n is the conditioning node of some e ∈ allCond}
    for each c ∈ condNodes do                     // For each conditioning node
        edges ← {e ∈ allCond | conditioning node of e is c}   // Collect its edges
        minw ← |min{weight(e) : e ∈ edges)}|      // Find the lowest weight
        sourceEdges ← {}
        for e ∈ edges do
            d ← reversed e with added weight minw // Non-negative distances
            sourceEdges ← sourceEdges ∪ d
        end
        LimitedDijkstra (G, D, C, sourceEdges, current, minw)
                          // Returns a set of reachable nodes and their distances
        for all nodes n ≠ c reached by LimitedDijkstra do   // Guarantees B ≠ D in D3
            e ← cond. edge (n → current), weight Dist (n) - minw
            if e is a tightening then
                add e to G
                apply D8 and D9 to e
            end
        end
    end
    return

function ProcessNegReq(EDG G, DDG D, TCGraph C, Node current)
                                                  // Indirectly applies D6 and D7
    edges ← IncomingNegReq(current, G)
    minw ← |min{weight(e) : e ∈ edges)}|          // Find the lowest weight
    sourceEdges ← {}
    for e ∈ edges do
        d ← reversed e with added weight minw     // Non-negative distances
        sourceEdges ← sourceEdges ∪ d
    end
    LimitedDijkstra (G, D, C, sourceEdges, current, minw)
                          // Returns a set of reachable nodes and their distances
    for all nodes n reached by LimitedDijkstra do
        e ← req. edge (n → current) of weight Dist (n) - minw
        if e is a tightening then add e to G
    end
    return

function ProcessPosReq(EDG G, Node current)   // Directly applies D1, D4 and D5
    for each e ∈ IncomingPosReq(current, G) do
        apply derivation rule D1, D4 and D5 with e as focus edge
        for any derived edge d do
            if d is a conditional edge then
                apply derivations D8-D9 with d as focus edge
            end
            for each derived tightening do    // This could be d or a derived
                add tightened edge to G       // requirement edge, or both
            end
        end
    end
    return
```

Algorithm 8: The Limited Dijkstra Function

```
function LimitedDijkstra(EDG G, DDG D, TCGraph C, Edge Set sourceEdges,
                         Node cur, Number maxDist)
    for each node n in D do                                        // Initialization
        Dist_cur[n] ← ∞
    end
    toVisit ←Priority queue sorted on distance to cur
    for each edge e in sourceEdges do
        Dist_cur[Target(e)] ← Weight(e)          // One of these will have distance 0
        Add Target(e) to toVisit
    end
    visited ← {}
    while toVisit is not empty do
        visiting ← dequeue from toVisit
        visited ← visited ∪ {visiting}
        if Dist_cur[visiting] < maxDist then
            for each outgoing edge e from visiting in D do
                p ← Target(e)
                if p=cur then continue                  // We don't follow paths into cur
                if Dist_cur[p] > Dist_cur[visiting] + Weight(e) then
                    Dist_cur[p] ← Dist_cur[visiting] + Weight(e)
                    Update key for p in toVisit
                    // Is a new ordering detected?
                    if the edge (p, cur) ∉ C and Dist_cur[p] < maxDist then
                        if p ∈ processing then
                            return visited, Dist          // Negative cycle detected
                        end
                        if p ∉ finished then  ProcessNode (G, D, C, p)
                    end
                end
            end
        end
    end
    return visited, Dist
```

the TCGraph to all TCGraph successors of *current*. This simple algorithm is enough to guarantee that the transitive closure is found. The complexity is within $O(n^2)$ since there are $O(n)$ incoming edges and sources, for which at most $O(n)$ time is needed to find its predecessors. Connecting these $O(n)$ predecessors to the $O(n)$ successors of *current* takes at most $O(n^2)$ time.

Dijkstra Distance Graph Handling. The last major point of modification is due to the recursive nature of E2IDC. Because of this the source edges used in the Dijkstra calculations are not added to the DDG. This is a necessity since by recursion several Dijkstra calculations may run in parallel. Instead, whenever a new Dijkstra calculation is started, the first step is done before entering the iterative loop. E2IDC initializes the distances to all nodes reachable from the source before starting the iterations. This makes it possible for concurrent Dijkstra calculations to progress without adding these special edges. In order to allow several Dijkstra's in parallel, there is a need to index the node distances by the *current* node. In the LimitedDijkstra function the index is shown in the subscript *current* to the `Dist` array. The LimitedDijkstra function needs some clarification. The function will continue to add nodes to the queue as long as the distance of the visiting node is less than $maxDist$. The reason for this

is that once the distance exceeds $maxDist$ any derived edges will be positive and hence further derivations from these are not possible. There are also two special cases that needs to be explained. We prevent *cur* from being added to $toVisit$. Any path into *cur* corresponds to a derivation of a loop, i.e. an edge from *cur* to *cur*. Such loops can be disregarded if positive and leads to non-DC if negative. We prevent it from causing loops in the recursion. The fact that the STNU is non-DC in these cases is caught be the local squeeze at the point just before creation of the loop edge was possible. The last special case is the one mentioned when the *processing* set was introduced. Detecting if a new order is found which involves a node in the *processing* set. This means that there must be a negative loop and the STNU is non-DC.

A final small modification is that there is no need to process the source of an externally tightened negative edge. The edge could only react with non-negative edges that are incoming to the source. There are two cases:

1. The source has incoming edges that are negative. These were then present in a previous increment, when the node was completely processed. Processing it again will not result in new non-negative edges.
2. If the source node only has incoming non-negative edges, then only D7 derivations are applicable, but these are isomorphic to D4 which will be applied when the target is processed.

We now show that discovering and processing another node while processing *current* does not affect the previous work done in *current*.

Lemma 4 (Pausing Lemma) *Suppose that while node n is being processed, it is discovered (type 2) that node m is after n. Suppose also that m is the first such node discovered while processing n. Then the Dijkstra calculations made before m was discovered are not affected by processing m intermediately before continuing with n. As a result the processing of n can "pause" until m is completely processed.*

Proof If m was already in *finished*, it would not need to be processed and the lemma holds. We now assume that it was not yet completely processed. There are two ways in which performed Dijkstra calculations could become obsolete by processing a later node:

1. A tighter distance towards m is found which later propagates to replace existing distances to nodes already visited by Dijkstra.
2. Edge weights between nodes that were already visited by Dijkstra shrinks, resulting in shorter distances from n.

In the first case note that the further along the Dijkstra calculation goes, the farther the distances become, i.e. the more positive the weights become. This means that the tightest replacement edge for the $n \leftarrow m$ edge is a tighter negative $n \leftarrow m$ edge that was derived when processing m. Recall that the only way of deriving an edge which targets an earlier node is by the interaction of a non-negative edge followed by a negative edge. In this case the negative

edge is the $n \leftarrow m$ edge. Therefore, any derivation results in an edge with a weight that consists of a non-negative value added to the weight on the $n \leftarrow m$ edge. This means that the weight of the derived edge cannot be tighter than the existing edge and the derived edge will be discarded.

In the second case an important aspect of the requirement in the lemma is that m must be the *first* node for which a new order is found. This means that any distance calculated by Dijkstra prior to finding m only involved nodes that were known to be after n. We can therefore apply a similar reasoning to that which is done in the Requirement Lemma and conclude that these nodes cannot be added to *todo* at a later time. As in the lemma this would lead to a contradiction regarding the possibility of choosing n for processing. Since the nodes will not be processed later they cannot receive tighter incoming edges during this increment. Therefore, any Dijkstra calculation over these edges is still valid after the recursive processing of m.

We see in both cases that the Dijkstra distances cannot be compromised by the intermediate processing of m. □

Note that ProcessNode can be seen as a sequence of Dijkstra calculations: First each conditioning node of incoming conditional edges requires a separate Dijkstra calculation and after this the incoming negative requirement edges requires an additional Dijkstra calculation. The pausing lemma states that it does not matter if a new ordering is found during the Dijkstra calculations. They can each be paused and resumed to complete the calculation. The lemma focuses on one such calculation. If a discovery of a new ordering takes place at a certain Dijkstra calculation then all Dijkstra calculations carried out before with the current focus node still remain valid. To see this assume that with n as *current*, ProcessCond processed the conditional edges conditioned on nodes $A_1, \ldots, A_n$ and when calculating the distances for conditioning node A_{n+1} a new ordering is discovered. Let m be the source node of the negative edge responsible for the new ordering. The fact that the ordering was only discovered at this point means that any edge derived by ProcessCond for the previous conditioning nodes which has m as source has non-negative weight. Therefore, there can be no later interaction in this increment from edges targeting m with the previously derived edges from ProcessCond and hence the distances previously calculated by ProcessCond are valid. All nodes from which negative edges were derived, or existed previously, during the prior conditional processings must have been processed before the current focus node was selected as *current* (same reasoning as in both proofs) and therefore they cannot receive additional incoming edges which could affect the so far executed Dijkstra calculations.

The lemma shows that it is safe to recursively handle any found ordering and then continue as if that ordering was in fact known when processing started. Therefore, the Dijkstra calculations for a particular node are only run in its entirety once before the node is finished. This gives the final result of the paper.

Corollary 2 (E2IDC Complexity) *The run-time of E2IDC is $O(n^3)$ in the worst case.*

Proof The worst case time complexity when no new order was found by EIDC is $O(n^3)$. Since this amounts exactly to the work done by E2IDC, even when new orderings are handled, the latter is $O(n^3)$. □

9 Related Work and Conclusion

In this paper we presented the Efficient^{2}IDC version of EIDC. It is an improvement which reaches a true $O(n^3)$ worst case per call, as compared to EIDC's worst case of $O(n^4)$. The result was reached through an in-depth analysis of which parts of the work done by EIDC that could be saved in case a new ordering was detected. The EIDC algorithm always remake Dijkstra calculations in case a node was re-processed. By recursively processing nodes until they are completely processed we prove that an $O(n^3)$ bound is attained.

We wish to clarify the relation between this work and that of Morris [7], which presents an algorithm for verification of dynamic controllability in a full STNU. Throughout our work, the main focus has been incremental verification. Though EIDC and Morris' algorithm have similar complexity results and share certain concepts, they were developed independently and the key ideas underlying EIDC were submitted and finalized before the publication of Morris' paper. That said, the recursion part of E2IDC was applied after Morris' paper. The key contribution of E2IDC is an improvement of the previous EIDC *incremental* DC verification algorithm, which is based on different graph representation, derivation rules and underlying theory than Morris' algorithm.

Other recent related work includes the Timed Game Automata (TGA) approach which allows inclusion of other aspects into STNUs, such as resource constraints and choice [2,3]. This approach works on a smaller scale and does not exploit the inherent structure of STNUs as distance graphs. Therefore it is more useful in networks that are small in size but involve additional capabilities which cannot be handled by pure STNU algorithms.

Acknowledgements 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

1. M.A. Bender, J.T. Fineman, S. Gilbert, and R.E. Tarjan. A new approach to incremental cycle detection and related problems. *arXiv preprint arXiv:1112.0784*, 2011.
2. Amedeo Cesta, Alberto Finzi, Simone Fratini, Andrea Orlandini, and Enrico Tronci. Analyzing flexible timeline-based plans. In *Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 471–476. IOS Press, 2010.

3. Alessandro Cimatti, Luke Hunsberger, Andrea Micheli, and Marco Roveri. Using timed game automata to synthesize execution strategies for simple temporal networks with uncertainty. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, July 27 -31, 2014, Québec City, Québec, Canada.*, pages 2242–2249, 2014.
4. Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.
5. Luke Hunsberger. Fixing the semantics for dynamic controllability and providing a more practical characterization of dynamic execution strategies. In *Temporal Representation and Reasoning, 2009. TIME 2009. 16th International Symposium on*, pages 155–162. IEEE, 2009.
6. Paul Morris. A structural characterization of temporal dynamic controllability. In *Proceedings of the 12th international conference on Principles and Practice of Constraint Programming*, pages 375–389. Springer-Verlag, 2006.
7. Paul Morris. Dynamic controllability and dispatchability relationships. In *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, pages 464–479. Springer, 2014.
8. Paul Morris and Nicola Muscettola. Temporal dynamic controllability revisited. In *In Proceedings of the 20th National Conference on Artificial Intelligence (AAAI-2005*, 2005.
9. Paul Morris, Nicola Muscettola, and Thierry Vidal. Dynamic control of plans with temporal uncertainty. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 494–499, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.
10. Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental dynamic controllability revisited. In *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS)*, 2013.
11. Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Classical Dynamic Controllability Revisited: A Tighter Bound on the Classical Algorithm. In *Proceedings of the 6th International Conference on Agents and Artificial Intelligence (ICAART)*, pages 130–141, 2014.
12. Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. EfficientIDC: A Faster Incremental Dynamic Controllability Algorithm. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS)*, 2014.
13. Mikael Nilsson, Jonas Kvarnström, and Patrick Doherty. Incremental Dynamic Controllability in Cubic Worst-Case Time. In *Proceedings of the 21th International Symposium on Temporal Representation and Reasoning (TIME)*, 2014.
14. Julie A. Shah, John Stedl, Brian C. Williams, and Paul Robertson. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In Mark S. Boddy, Maria Fox, and Sylvie Thibaux, editors, *Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 296–303. AAAI Press, 2007.
15. John L. Stedl. Managing temporal uncertainty under limited communication: A formal model of tight and loose team coordination. Master's thesis, Massachusetts Institute of Technology, 2004.
16. I. Tsamardinos. Reformulating temporal plans for efficient execution. Master's thesis, University of Pittsburgh, 2000.
17. Thierry Vidal and M. Ghallab. Dealing with uncertain durations in temporal constraints networks dedicated to planning. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI)*, pages 48–52, 1996.