An Efficient Decremental Algorithm for Simple Temporal Networks

Yehor Kozyr[1][0009−0001−4445−7833], Wéam Aridi[2][0000−0001−6418−6332], and Neil Yorke-Smith[1][0000−0002−1814−3515]

[1] Delft University of Technology, The Netherlands
[2] University of Cambridge, United Kingdom
n.yorke-smith@tudelft.nl

Abstract. Algorithms for solving simple temporal networks (STN) are expensive in large-scale environments. This matters in use cases such as practical AI planning systems. When a small number of constraints are updated, we wish to recompute incrementally. Prior art considers incremental computation when constraints are added or tightened. This paper presents the first algorithm for incremental computation when constraints are removed or loosened. The presented DPPC algorithm is based on the refined concept of a support graph. Experimental results on a benchmark library of STN instances find that DPPC outperforms a state-of-the-art non-incremental algorithm in over 95% of cases.

Keywords: temporal reasoning · simple temporal network · support graph · incremental algorithms

1 Introduction

Planning and scheduling is a long-standing area of AI [6]. The core scheduling problem at the centre of notable deployed AI systems is known as the Simple Temporal Network (STN) [1]. The STN is a graph representation with linear constraints. There is a long line of work on the STN and variants [4, 7]. The main inference task over STNs is checking or enforcing consistency of the constraints.

A state-of-the-art algorithm for consistency of STNs is P3C [9]. P3C can be thought of a specific implementation of partial path consistency for the constraint class embodied by STNs. P3C is a monolithic algorithm. By contrast, in practical uses of STNs – such as in planning when the underlying STN is frequently updated – we wish to recompute consistently *incrementally*, rather than monolithically as with P3C [3, 5, 11, 2, 10].

An incremental algorithm for STN constraint addition or tightening was developed by ten Thije [12]. It is based on the concept of a STN's *support graph*. The algorithm takes advantage of P3C but applies it to a judicious subset of the graph. The authors demonstrate that such an approach is advantageous in case of incremental updates – updates that restrict already existing restrictions. Another incremental algorithm is by Micheli [8]; it also uses an auxiliary graph structure, the δ-STN.

For *decremental* updates – such as updates that loosen existing restrictions – the idea of a support graph algorithm was proposed also by ten Thije [12] but not tested experimentally. The research gap is the question whether ten Thije [12]'s theoretical algorithm represents an effective decremental approach. This paper address exactly this question. The work we present stems from the bachelor's thesis of the first author.

In this work, we make two contributions to the literature. First we present improvements to the theoretical DPPC algorithm of ten Thije [12], sharply reducing the number of calls in updating the support graph. Second, we undertake the first empirical evaluation of decremental solving of STNs, in particular comparing the improved DPPC to P3C. We benchmark on structured STNs arising from hierarchical task network planning, and on random scale-free graphs. Results show the dominance of DPPC with a lower run-time in 96.5% of cases and a lower average run-time.

The remainder of the paper is structured as follows. In Section 2 we introduce the STN and the idea of a support graph, as well as the algorithm and pseudocode for DPPC. In Section 3 we explain the methodology behind the implementation and evaluation of DPPC, as well as our improvements to the algorithm. In Section 4 we present results obtained during experiments comparing DPPC and P3C algorithms. Section 5 concludes the paper.

2 Background and Related Work

2.1 Simple temporal network

A *simple temporal network* consists of a set $X = \{x_1, x_2, \ldots, x_n\}$ representing a set of n events and a set C of m constraints [4]. Each constraint $c_{i\to j} \in C$ has a weight $w_{i\to j}$, which represents an upper bound on the time passed between events x_i and x_j, and can be rewritten in the form $x_j - x_i \leq w_{i\to j}$. Additionally, it is easy to show that $c_{j\to i}$ contains information about the lower bound on the time difference between x_i and x_j: as $x_i - x_j \leq w_{j\to i}$, it follows that $x_j - x_i \geq -w_{j\to i}$. Given that we can convert weights to lower and upper bounds of STNs, we can model STNs as graphs, where in case of $c_{i\to j}$ there is a directed edge from x_i to x_j with a weight $w_{i\to j}$. For a full introduction to STNs and their extensions, we refer to Dechter et al. [4] and subsequent literature.

The main inference task over a STN is to determine whether the set of constraints is a priori consistent. Further, the constraints of a STN can be tightened to a so-called *minimal network* [4], if the graph is not already minimal (Figure 1). We will refer to this process of checking consistency and (if consistent) obtaining the minimal network, as 'solving' a STN. P3C is an example of such a solving algorithm [9]. Further, we denote as E a set of edges of the original graph over vertices X with weights $w_{i\to j}$, and as E_{min} a set of edges of the minimal graph with (minimal) weights $\omega_{i\to j}$.

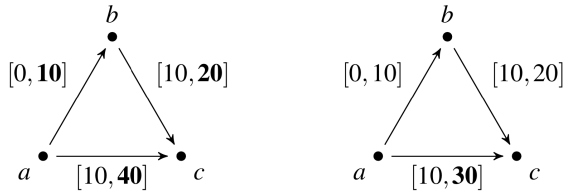


Fig. 1: Example of a STN and how constraints can affect other constraints in the tightened version of the STN. Reproduced from ten Thije [12].

2.2 Support graph

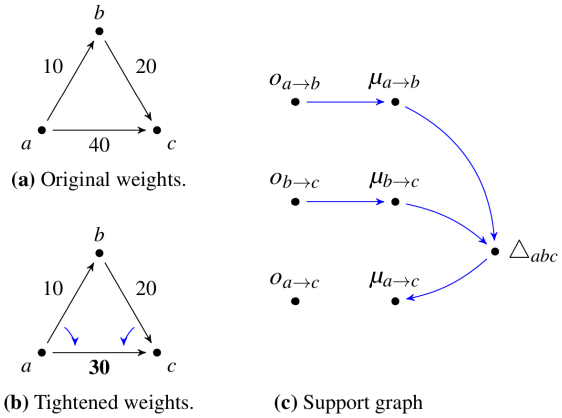
Incremental solving of STNs is based on the idea of a *support graph* [12]. This is a data structure computed and maintained separately from computing the edges E_{min} of the STN's minimal graph. The reason for a separate data structure is that it allows us to determine the support for *every* minimal weight that is computed.

The intuition is that every minimal weight is formed by either 1) the sum of other minimal weights of the graph or 2) a single edge that connects two nodes, and is the shortest path from one node to another. This allows us to reduce the set of edges that need to be recalculated. This is useful when a new constraint is added to the STN, or an existing constraint is narrowed such as by events occurring in a planning system: for instance from $[10, 30]$ to $[10, 20]$.

The support graph can be thought of as a structure representing how different edges affect the minimum weight to get from one vertex to another. Each edge in the support graph going from vertex a to b shows some effect of vertex a onto b. Figure 2 depicts a STN, its tightened minimal network, and the corresponding support graph.

In more detail, the support graph contains vertices of three types, each with a different semantic:

1. A vertex of type $o_{a\to b}$ represents the weight of the edge that connects vertices a and b in the original graph.
2. A vertex of type $\mu_{a\to b}$ represents the minimum weight of the path connecting vertices a and b in the original graph. A direct edge in a support graph from $o_{a\to b}$ to $\mu_{a\to b}$ represents that the edge between a and b in the original graph is a path with minimum total weight from a to b.
3. In order to describe paths with more than one edge in them, the support graph has a third type of vertices of type $\triangle_{abc}$. These vertices cannot exist without incoming or outgoing edges: if there is a $\triangle_{abc}$, it is guaranteed that there exists an edge from $\mu_{a\to b}$ to $\triangle_{abc}$, an edge from $\mu_{b\to c}$ to $\triangle_{abc}$, and an edge from $\triangle_{abc}$ to $\mu_{a\to c}$. Such a combination of vertices represents the fact that $\omega_{a\to c} = \omega_{a\to b} + \omega_{b\to c}$. In such a scenario, we say that minimum weights of paths from a to b and from b to c support the minimum weight



(a) Original weights. **(b)** Tightened weights. **(c)** Support graph

Fig. 2: Example of a STN (a), its tightened representation (b), and support graph (c) based on the tightened weights. Reproduced from ten Thije [12].

Algorithm 1 Construct-Support-Graph

Input: (1) A PPC STN $S = \langle V, E, E_{min} \rangle$ where each constraint $c_{i\to j} \in C$ is annotated by its original weight $w_{i\to j}$ (in E) and its minimum weight $\omega_{i\to j}$ (in E_{min}). (2) A simplicial elimination ordering d.

Output: The weight support graph D for S.

1: $V', E' \leftarrow \emptyset$
2: **for all** $(u, v) \in E$ **do**
3: add $o_{u\to v}$ to V'
4: **for all** $(u, v) \in E_{min}$ **do**
5: add $\mu_{u\to v}$ to V'
6: mark $\mu_{u\to v}$
7: $D \leftarrow (V', E')$
8: **call** Update-Support-Graph(S, d, D)
9: **return** D

of a path from a to c, and such path can be represented as two connected paths – $a \to b$ and $b \to c$.

2.3 DPPC algorithm

Decremental reasoning concerns updates to the STN that loosen or delete one or more constraints. For example, from $[10, 30]$ to $[5, 35]$. The key observation underlying the original DPPC algorithm is that, if a constraint is loosened, some of such edges may no longer provide support to a minimal weight, as some other combination might be the shortest path now. Note that, in terms of the STN's graph, although such updates are considered decremental, as they weaken the constraints, they actually increase the weight of edges (Figure 2).

Algorithms 1–5 provide the pseudocode of the original, theoretical DPPC, describing all parts of the support graph lifecycle: its construction and the execution of updates upon it [12].

Algorithm 2 Support-DPPC

Input: (1) A graph $S = \langle V, E\rangle$ and a simplicial elimination ordering d. (2) A weight support graph $D = \langle V', E'\rangle$. (3) New weight $w'_{i\to j} > w_{i\to j}$ for the edge (i, j). (4) Edge $(a, b) \in E$.
Output: Updated graphs S and D.

$w_{a\to b} \leftarrow w'_{a\to n}$
if $(o_{a\to b}, \mu_{a\to b}) \notin E'$ **then**
    return
remove $(o_{a\to b}, \mu_{a\to b})$ from E'
if $N_{in}(\mu_{a\to b}) \neq 0$ **then**
    return
$V^* \leftarrow$ Affected-Endpoints$(S, D, (a, b))$
$V^+ \leftarrow$ Shared-Neighbours$(S, D, (a, b))$
call P3C$(S_{V^+},\ d_{V^+})$
call Update-Support-Graph$(S_{V^+},\ d_{V^+},\ D)$

Algorithm 3 Update-Support-Graph

Input: (1) A PPC STN $S = \langle V, E, E_{min}\rangle$ where each constraint $c_{i\to j} \in C$ is annotated by its original weight $w_{i\to j}$ (in E) and its minimum weight $\omega_{i\to j}$ (in E_{min}). (2) A simplicial elimination ordering $d = (v_n, v_{n-1}, ..., v_1)$. (3) A weight support graph $D = \langle V', E'\rangle$ in which no edges are adjacent to any marked minimum node.
Output: The weight support graph D for S.

for all $(u, v) \in E$ **do**
    if $\omega_{u\to v} = w_{u\to v}$ and $\mu_{u\to v}$ is marked **then**
        add $(o_{u\to v}, \mu_{u\to v})$ to E'
for $k \to 1$ **to** n **do**
    for all $i, j < k$ **such that** $\{v_i, v_j\}, \{v_j, v_k\} \in E$ **do**
        call find-support$(i,\ k,\ j)$
        call find-support$(k,\ j,\ i)$
        call find-support$(i,\ j,\ k)$
for all $(u, v) \in E_{min}$ **do** clear mark on $\mu_{u\to v}$
return D

Procedure 4 find-support

if $\mu_{x\to y}$ is marked and $\omega_{x\to y} = \omega_{x\to z} + \omega_{z\to y}$ **then**
    add $\triangle_{xzy}$ to V'
    add $(\mu_{x\to z}, \triangle_{xzy})$, $(\mu_{z\to y}, \triangle_{xzy})$, and $(\triangle_{xzy}, \mu_{x\to y})$ to E'

Procedure 5 Affected-Endpoints

Input: (1) A PPC STN $S = \langle V, E, E_{min}\rangle$ where each constraint $c_{i\to j} \in C$ is annotated by its original weight $w_{i\to j}$ (in E) and its minimum weight $\omega_{i\to j}$ (in E_{min}). (2) A weight support graph $D = \langle V', E'\rangle$. (3) Updated edge $a \to b$
Output: Set of all affected vertices of μ-type.

initialize $Q \leftarrow \emptyset$
enqueue $\mu_{a\to b}$ in the queue Q
while Q is not empty **do**
    dequeue $\mu_{u\to v}$ from Q
    if $N_{in}(\mu_{u\to v}) = \emptyset$ **then**
        mark $\mu_{u\to v}$
        add u and v to V^*
        reset $\omega_{u\to v}$ to $w_{u\to v}$
        for all $\triangle_{xyz} \in N_{out}(\mu_{u\to v})$ **do**
            enqueue $\mu_{x\to z}$ in Q
            remove $\triangle_{xyz}$ and edges attached to it from D
return V^*

Procedure 6 Improved Update-Support-Graph

Input: (1) A PPC STN $S = \langle V, E, E_{min} \rangle$ where each constraint $c_{i \to j} \in C$ is annotated by its original weight $w_{i \to j}$ (in E) and its minimum weight $\omega_{i \to j}$ (in E_{min}). (2) A simplicial elimination ordering $d = (v_n, v_{n-1}, ..., v_1)$. (3) A weight support graph $D = \langle V', E' \rangle$ in which no edges are adjacent to any marked minimum node.
Output: The weight support graph D for S.

1: **for all** marked $\mu_{u \to v} \in V'$ **do**
2: **if** $\omega_{u \to v} = w_{u \to v}$ **then**
3: add $(o_{u \to v}, \mu_{u \to v})$ to E'
4: unmark $\mu_{u \to v}$
5: **for all** marked $\mu_{u \to v} \in V'$ **do**
6: **for all** $j \neq v$ and $(u, j) \in E_{min}$ **do**
7: **call** FIND-SUPPORT(u, v, j)
8: clear all markings
9: **return** D

DPPC – called Support-DPPC in ten Thije [12] – works by identifying and marking vertices of type $\mu_{a \to b}$ that could have been affected by changing the weight. After that, the P3C algorithm is invoked on an identified subset of vertices. After we have found the new minimal weights for this subset, we need to restore support for them. This can be done through $o_{a \to b}$ or through other μ-type vertices, forming a $\triangle$-type node that would have an outgoing edge to the μ vertex of interest.

3 Developing an Efficient Algorithm

We improve the proposed (theoretical) algorithm DPPC by introducing nuanced conditions for initializing the iteration of the algorithm's key loop.

An important omission from the original DPPC pseudocode [12] is the following. While the loops at lines 4 and 5 in the Update-Support-Graph method (Algorithm 3) perform a check for constructing support edges, it is not necessary to check that many edges. As a consequence of the use of the Affected-Endpoints procedure (Algorithm 5), the only vertices that require new support edges are the marked vertices.

In more detail, the original Update-Support-Graph method (Algorithm 3) checks all possible combinations of paths $u \to j \to v$ and checks whether it is possible to construct a support for $\mu_{u \to v}$, but the check of whether $\mu_{u \to v}$ is marked is conducted only within find-support method (Algorithm 4). The consequence is redundant calls to the find-support method. Instead, by iterating over only marked vertices $\mu_{u \to v}$, we can check all vertices j such that there exists a path $u \to j \to v$, and reduce the number of calls to the find-support method as the check for marked would be done outside of the loop. This leads to skipping over combinations of u and v that do not need a new support edge, but that however were previously checked in the original pseudocode.

Therefore, a combination of loops to restore supports and the find-support method can be simplified to the procedure in Algorithms 6 and 7.

Procedure 7 Improved find-support

if $\omega_{x\to y} = \omega_{x\to z} + \omega_{z\to y}$ **then**
add $\triangle_{xzy}$ to V'
add $(\mu_{x\to z}, \triangle_{xzy})$, $(\mu_{z\to y}, \triangle_{xzy})$, and $(\triangle_{xzy}, \mu_{x\to y})$ to E'

4 Experimental Results

Experiments were conducted on a single core of an Intel Xeon Platinum 8358P CPU running at 2.60GHz and with 3.5 GB of RAM, running Red Hat Enterprise Linux 9.4. The DPPC algorithm was built in the language Java on top of the existing codebase of Planken [9, 10].

We look first at the relative performance of P3C and DPPC on single graphs of various types, and then at their scaling performance as the STN grows in number of vertices. Preliminary experiments (not reported) found that the unimproved DPPC using the original Algorithm 3 instead of the improved Algorithm 6 was consistently slower – up to orders of magnitude slower, in fact – than simply recomputing the whole STN using P3C. This was due to the overhead of maintaining the support graph.

4.1 Instance generation

Given a STN, we need a means to generate (decremental) updates to it. We settled on a proportional approach. This approach multiplies the weight of randomly-selected edges by a given coefficient according to a formula $|w| \cdot scale + w$, allowing only incremental updates to be generated. This way, we could easily generate updates with different levels of scaling, allowing us to analyse the behaviour of DPPC in different scenarios with different levels of disruption of the minimal weights. We call parameter w the 'update generator constant'.

Instances of two types of STN graphs formed the benchmark set: hierarchical task network (HTN) graphs, and scale free (SF) graphs.

The HTN-derived STN instances represent temporal sub-problems from a class of knowledge-based planning [13]. They were generated using an HTN generator from the library by Planken et al. [9] with parameters of branch depth in a range of $[3, 8]$, number of branches in a range of $[3, 14]$, landmark ratio of 0.2, and probability of SR-edge of 0.5. This almost exactly replicates the setup in (incremental) experiments by ten Thije [12], slightly increasing maximum branch depth and number of branches to adjust for a bigger number of tasks. The scale-free graphs (SF) were generated with a degree of 3.

4.2 Performance comparison on single graphs

With the first experiment we seek to test the main hypothesis regarding the algorithm: since DPPC applies the P3C algorithm on the reduced set of vertices, it should perform better than P3C in most scenarios where there is a modest amount of change in the STN.

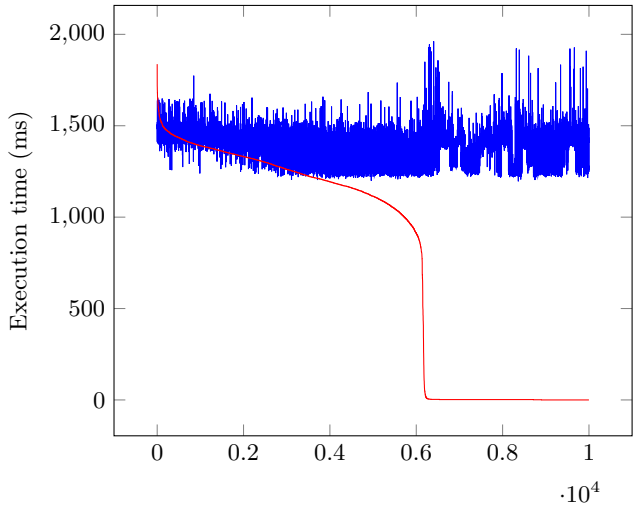


Fig. 3: Comparison of performance of DPPC (red) and P3C (blue) on an HTN graph, 0.5 generator constant, 3300 vertices. The horizontal axis corresponds to the index in the sorted list of measurements by the time of DPPC execution.

To test this hypothesis we conducted a series of runs on graphs of the same size to explore how DPPC performs on them in comparison to P3C. Each run is parameterised by the combination of the update generator constant (see above), the type of the graph (HTN, SF), and the graph size (3300 vertices for HTN graphs, 1500 vertices for SF graphs). For each combination, 10 graphs were generated; on each graph, 10 sets of 100 updates each were generated and executed. Each set contains 100 entries indicating the edge and the weight change made to the specified edge. For each entry, the weight in the graph is changed and followed by the execution of both algorithms recalculating minimum weights. The time taken for DPPC and P3C execution is measured and recorded.

In order to account for initialisation effects (Java initialisation, garbage collection, cache misses), for each update set the first 10 updates were used as a warmup, after which the graph was reset to its original state and the entire update set was executed.

Figure 3 and Table 1 show that DPPC outperforms P3C in most configurations. While sometimes DPPC's total execution times are higher than that of P3C, DPPC still outperforms P3C on average. This holds for both tested types of graphs (hierarchical task network and scale-free graphs). Table 2 reports the average and standard deviations of runtimes. The high standard deviation of DPPC is discussed in the sequel.

The results reported include updates to the graph where DPPC could stop immediately, after verifying that the an update would not result in a change in minimal weights. Table 3 indicates how many updates were discarded by DPPC, a feature which allowed DPPC to gain a significant advantage in execution times.

Table 1: Number of times DPPC has a strictly lower runtime than P3C, 10000 measurements per combination of graph type and generator constant.

Generator constant	HTN 3300 vertices	SF 1500 vertices
0.1	9565 (95.65%)	9711 (97.115%)
0.5	9548 (95.48%)	9749 (97.49%)
1	9518 (95.18%)	9768 (97.68%)
2	9567 (95.67%)	9758 (97.58%)

Table 2: Average and standard deviation of run time of DPPC and P3C algorithms on HTN with 3300 vertices, 10000 measurements per generator constant.

Generator constant	DPPC [ms] avg ± std	P3C [ms] avg ± std
0.1	830.12 ± 556.66	1333.49 ± 88.15
0.5	863.31 ± 585.93	1390.40 ± 95.60
1	883.96 ± 599.32	1422.62 ± 94.74
2	831.33 ± 566.00	1350.95 ± 90.59

As an ablation in the data, Table 4 reports the same as the earlier Table 1 but excluding those runs where DPPC immediately disregarded an update.

Table 4 demonstrates the advantage of DPPC over P3C, making it a preferable alternative in most of the scenarios where updates change the minimal weight: in 94.9% on HTN graphs and in 96.7% on SF among conducted updates in this experiment.

Figure 4 shows the distribution of differences in the run time between P3C and DPPC on the same updates.

Figure 5 likewise reports the same distribution but excluding entries with an early exit of DPPC. Both Figures 4 and 5 demonstrate a strong advantage of DPPC over P3C. The peak on the right – the closest to 0 – can be attributed to 'regular' behaviour of DPPC when early exit is not triggered, showing how a

Table 3: Proportion of times when DPPC stopped early in the experiment, 10000 measurements per combination of graph type and generator constant.

Generator constant	HTN 3300 vertices	SF 1500 vertices
0.1	9.84%	19.95%
0.5	11.09%	21.87%
1	12.76%	23.31%
2	14.73%	25.09%

Table 4: Proportion of measurements in which DPPC has a strictly lower runtime than P3C, 10000 measurements conducted per combination of graph type and generator constant experiments, excludes measurements with early exit.

Generator constant	HTN 3300 vertices	SF 1500 vertices
0.1	95.18%	96.39%
0.5	94.92%	96.79%
1	94.48%	96.97%
2	94.92%	96.77%
All	94.87%	96.73%

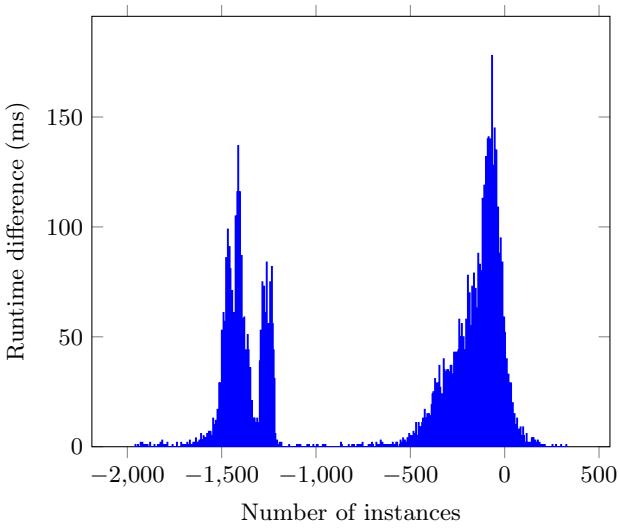


Fig. 4: Distribution of the difference in runtime between DPPC and P3C in the experiment with 0.5 generator constant, 3300 vertices HTN, including entries with early exit.

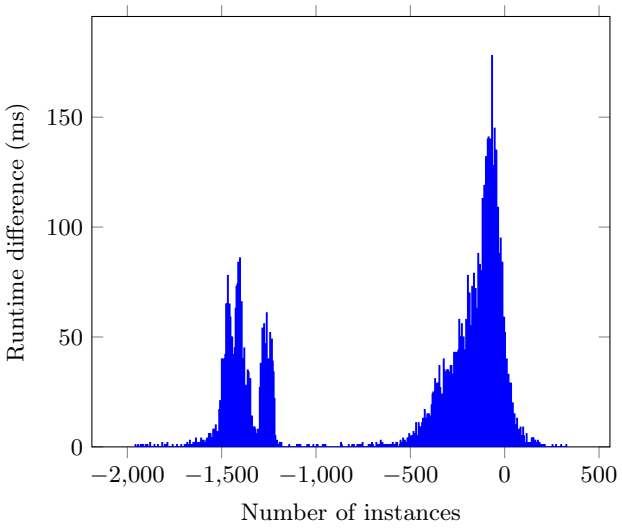
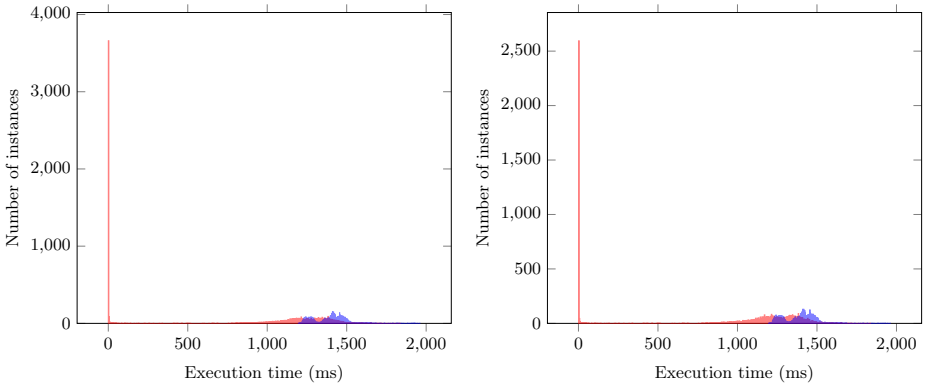


Fig. 5: Distribution of the difference in runtime between DPPC and P3C in the experiment with 0.5 generator constant, 3300 vertices HTN, excluding entries with early exit.

precomputed support graph helps identify a subset of the problem, reducing the number of edges needing a recalculation resulting in a lower runtime.

Variance of DPPC. The peaks on the left of the graphs were less expected; their presence explains the higher variance. Our original assumption was that the peaks are related to updates with an early exit, but it is evident from the figures that the peaks stay in the plots even when entries with early exits are removed. Figures 6a and 6b show distributions of the run-time of the algorithms, in the same experiment including and excluding early exit entries respectively.

Both figures exhibit a peak in their distributions, indicating that DPPC executes 'too fast' in some scenarios – execution times are less than 10ms while the average times is around 800 ms. While this at first seems anomalous, it appears to be expected. Due to the fact that DPPC executes on a smaller subset of vertices, its performance is heavily impacted by the size of the graph formed by affected vertices – the smaller the number of affected vertices and edges, the faster the execution would be. Thus these peaks in previous figures, are related to the size of the identified subset size. Figure 7 demonstrating the distribution of the number of affected vertices (size of V^+ set) supports this assumption, showing a peak of similar size around 0. These peaks also explain the high standard deviation in the run time of DPPC in Table 2.



(a) Including entries with early exit (b) Excluding entries with early exit

Fig. 6: Distribution of the runtime of DPPC (red) and P3C (blue): 0.5 generator constant, 3300 vertices HTN.

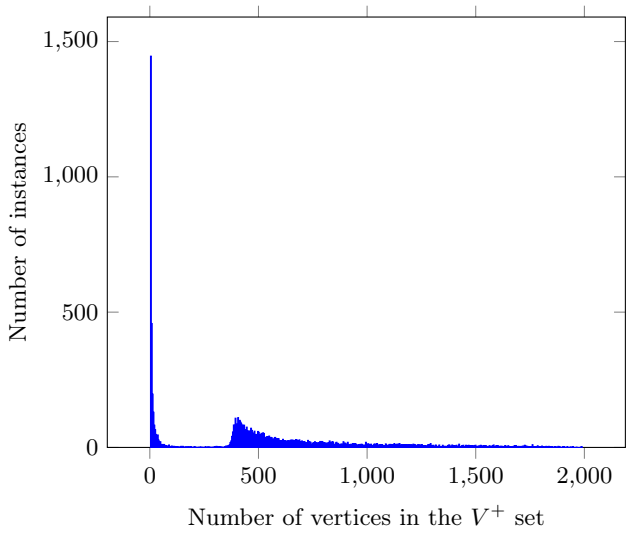


Fig. 7: Distribution of the size of V^+ in the experiment with 0.5 generator constant, 3300 vertices HTN, excludes entries with early exit.

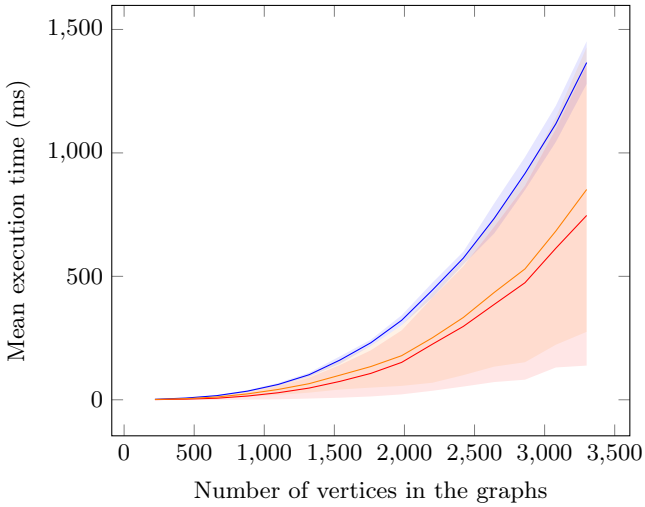


Fig. 8: Mean execution time with the growing size of HTN graph comparison of P3C (blue), DPPC (red) and DPPC excluding entries with early exit (orange), 2000 measurements per graph size. Filled region indicates standard deviation.

4.3 Performance comparison on different sizes of graphs

Thus far we have studied the relative performance of DPPC to P3C on the same STN instances. A second hypothesis is that DPPC should perform better than P3C as the size of the STN grows in terms of the number of vertices.

Since the main point of DPPC is to reduce the subset of vertices that need to be recalculated in comparison to P3C that runs on the entire graph, we expect to find empirical evidence that bigger graphs would benefit more in run time from DPPC over P3C. This is because the gap between the number of affected and unaffected vertices would increase on average.

To test this hypothesis, we conducted benchmarks on graphs of different sizes, in terms of number of vertices. For each size, five graphs were generated, and four sets of updates 100 each were used for each graph, generating 2000 measurements per the size of the graph. Graphs were generated using an HTN generator the same parameters described earlier.

Figure 8 shows that DPPC outperforms P3C, extending the gap in performance with growth in the number of vertices. The performance advantage of DPPC remains evident even when the early exit feature is disabled.

5 Conclusion and Outlook

This paper closed a gap in the literature between a theoretical approach to decremental reasoning over STNs, and an efficient and empirically verified algo-

rithm. Starting with the theoretical algorithm of ten Thije [12], we developed an improvement for the critical Update-Support-Graph component.

The presented DPPC algorithm was benchmarked against the state-of-the-art non-incremental algorithm P3C. Across the benchmark sets of STN instances, DPPC outperforms monolithic use of P3C on average, showing diminished performance only in less than 5% of the updates. Further, DPPC has a lower average execution time without considering automatically discarded updates. That is, without early exit DPPC still outperforms P3C. Additionally, the reason behind the run-time distribution DPPC was investigated and seems to correlate with the number of affected nodes. DPPC performs equally well on hierarchical task network graphs and scale-free graphs.

In future work, it would be interesting to see the effect of incremental and decremental solving combined, especially on practical STN instances arising, e.g. from planning systems [8, 10]. Second, the incremental (and potentially decremental extensions of) STN algorithms of ten Thije [12] and Micheli [8] have not, to our knowledge, been contrasted empirically. Lastly, the use of learned heuristics for incremental and decremental reasoning can be explored.

Acknowledgements The authors thank the BNAIC'24 reviewers for their suggestions. Thanks to L. Planken. Thanks also to F. Doolaard, D. Graur, T. Yue, B. Zablocki, and to S. van der Laan. This research was partially supported by TAILOR, a project funded by EU Horizon 2020 programme under grant number 952215.

References

[1] Ai-Chang, M., Bresina, J.L., Charest, L., Chase, A., Hsu, J.C., Jónsson, A.K., Kanefsky, B., Morris, P.H., Rajan, K., Yglesias, J., Chafin, B.G., Dias, W.C., Maldague, P.F.: MAPGEN: Mixed-initiative planning and scheduling for the Mars Exploration Rover mission. IEEE Intelligent Systems 19(1), 8–12 (2004), https://doi.org/10.1109/MIS.2004.1265878

[2] Cesta, A., Oddi, A.: Gaining efficiency and flexibility in the simple temporal problem. In: Proceedings of the Third International Workshop on Temporal Representation and Reasoning (TIME'96). pp. 45–50 (1996), https://doi.org/10.1109/TIME.1996.555676

[3] Coles, A., Coles, A., Fox, M., Long, D.: Incremental constraint-posting algorithms in interleaved planning and scheduling. In: Proceedings of the ICAPS'09 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling (COPLAS'09) (2009), https://icaps09.icaps-conference.org/workshops/ICAPS2009-WS2-proceedings.zip

[4] Dechter, R., Meiri, I., Pearl, J.: Temporal constraint networks. Artificial Intelligence 49(1), 61–95 (1991), https://doi.org/10.1016/0004-3702(91)90006-6

[5] Gerevini, A., Perini, A., Ricci, F.: Incremental algorithms for managing temporal constraints. In: Proceedings of the Eighth International Conference on Tools with Artificial Intelligence (ICTAI'96). pp. 360–365 (1996), https://doi.org/10.1109/TAI.1996.560477

[6] Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning - Theory and practice. Elsevier (2004)

[7] Hunsberger, L., Posenato, R.: Simple temporal networks: A practical foundation for temporal representation and reasoning. In: Proceedings of the Twenty-Eighth International Symposium on Temporal Representation and Reasoning (TIME'21). LIPIcs, vol. 206, pp. 1:1–1:5 (2021), https://doi.org/10.4230/LIPIcs.TIME.2021.1

[8] Micheli, A.: An efficient incremental simple temporal network data structure for temporal planning. CoRR abs/2212.07226 (2022), https://doi.org/10.48550/arXiv.2212.07226

[9] Planken, L., de Weerdt, M., van der Krogt, R.: P3C: A new algorithm for the simple temporal problem. In: Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS'08). pp. 256–263 (2008), http://www.aaai.org/Library/ICAPS/2008/icaps08-032.php

[10] Planken, L., de Weerdt, M., Yorke-Smith, N.: Incrementally solving STNs by enforcing partial path consistency. In: Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS'10). pp. 129–136 (2010), http://www.aaai.org/ocs/index.php/ICAPS/ICAPS10/paper/view/1447

[11] Ramalingam, G., Reps, T.W.: An incremental algorithm for a generalization of the shortest-path problem. Journal of Algorithms 21(2), 267–305 (1996), https://doi.org/10.1006/jagm.1996.0046

[12] ten Thije, J.O.A.: Towards a dynamic algorithm for the Simple Temporal Problem. Master's thesis, Delft University of Technology (2011), https://resolver.tudelft.nl/uuid:bb91b0cf-9ce5-4470-a326-b61885d34988

[13] Yorke-Smith, N.: Exploiting the structure of hierarchical plans in temporal constraint propagation. In: Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI'05). pp. 1223–1228 (2005), http://www.aaai.org/Library/AAAI/2005/aaai05-194.php