Incrementally Solving STNs by Enforcing Partial Path Consistency

Léon Planken and **Mathijs de Weerdt**
Delft University of Technology
{l.r.planken,m.m.deweerdt}@tudelft.nl

Neil Yorke-Smith
American University of Beirut and SRI International
nysmith@aub.edu.lb

Abstract

Efficient management and propagation of temporal constraints is important for temporal planning as well as for scheduling. During plan development, new events and temporal constraints are added and existing constraints may be tightened; the consistency of the whole temporal network is frequently checked; and results of constraint propagation guide further search. Recent work shows that enforcing partial path consistency provides an efficient means of propagating temporal information for the popular Simple Temporal Network (STN). We show that partial path consistency can be enforced *incrementally*, thus exploiting the similarities of the constraint network between subsequent edge tightenings. We prove that the worst-case time complexity of our algorithm can be bounded both by the number of edges in the chordal graph (which is better than the previous bound of the number of vertices squared), and by the degree of the chordal graph times the number of vertices incident on updated edges. We show that for many sparse graphs, the latter bound is better than that of the previously best-known approaches. In addition, our algorithm requires space only linear in the number of edges of the chordal graph, whereas earlier work uses space quadratic in the number of vertices. Finally, empirical results show that when incrementally solving sparse STNs, stemming from problems such as Hierarchical Task Network planning, our approach outperforms extant algorithms.

1. Introduction

Quantitative temporal constraints are essential for many real-life planning and scheduling domains (Smith, Frank, and Jónsson 2000). The *Simple Temporal Network* (STN) (Dechter, Meiri, and Pearl 1991) is a popular model for temporal events and simple temporal constraints among them. The central role of STNs in deployed planning systems (Laborie and Ghallab 1995; Bresina et al. 2005; Castillo, Fdez-Olivares, and O. Garca-Pérez 2006) makes efficient inference with STNs especially important.

During plan development, new events and temporal constraints are added and existing constraints may be tightened, and the consistency of the whole temporal network is frequently checked. Recent work has shown that enforcing *partial path consistency* provides an efficient means of propagating temporal information for an STN (Xu and Choueiry 2003; Planken, de Weerdt, and van der Krogt 2008), answering queries such as:

- Is the information represented by the STN consistent?
- How can the events be scheduled to meet all constraints?
- What are all possible times at which some event X_i could occur?
- What relation between two events X_i and X_j is implied by the current set of constraints?

These types of queries Dechter, Meiri, and Pearl (1991) identified as the most relevant when solving STNs.

This paper introduces a new algorithm that maintains partial path consistency when new constraints are added or existing constraints are tightened. Our method, *IPPC*, is based on the idea that, in order to maintain partial path consistency, the weights of edges in a chordal graph only need to be updated if at least one of the neighbours has an incident edge that is updated. Both theoretical and empirical results show that our new approach outperforms extant algorithms for incrementally solving sparse STNs, such as those generated from Hierarchical Task Network planning.

We first give the necessary definitions, followed by the details of our algorithm. We present proofs of correctness and upper bounds on run time and space. We next discuss alternative approaches and then empirically compare IPPC with two of the most efficient existing methods for incrementally solving STNs. Ideas for future work conclude the paper.

2. Preliminaries

A Simple Temporal Problem instance consists of a set $X = \{x_1, \ldots, x_n\}$ of time point variables representing events, and a set C of m constraints over pairs of time points, bounding the temporal difference between events. Every constraint $c_{i\to j}$ has a weight $w_{i\to j} \in \mathbb{R} \cup \{\infty\}$ corresponding to an upper bound on the difference, and thus represents an inequality $x_j - x_i \leq w_{i\to j}$. Two constraints $c_{i\to j}$ and $c_{j\to i}$ can be combined into a single constraint $c_{i\leftrightarrow j} : -w_{j\to i} \leq x_j - x_i \leq w_{i\to j}$ or, equivalently, $x_j - x_i \in [-w_{j\to i}, w_{i\to j}]$, giving both upper and lower bounds. An unspecified constraint is equivalent to a constraint with an infinite weight; therefore, if $c_{i\to j}$ exists and $c_{j\to i}$ does not, we have $c_{i\leftrightarrow j} : x_j - x_i \in (-\infty, w_{i\to j}]$.

Algorithm 1: IPPC

```
Input: A PPC STN $\mathcal{S} = \langle V, E\rangle$ with associated weights
       $\{w_{i\to j} \mid \{i,j\} \in E\}$ and a new constraint with
       weight $w'_{a\to b}$ for some $\{a,b\} \in E$
Output: INCONSISTENT if the new weight yields
        inconsistency; CONSISTENT if PPC has been
        reinstated on $\mathcal{S}$
1  if $w'_{a\to b} + w_{b\to a} < 0$ then return INCONSISTENT;        *
2  if $w'_{a\to b} > w_{a\to b}$ then return CONSISTENT;              *
3  forall $v \in V$ do
4      $D_{a\leftarrow}[v] \leftarrow \infty$; $D_{b\to}[v] \leftarrow \infty$;    *
5      COUNT$[v] \leftarrow 0$; TAGGED$[v] \leftarrow$ FALSE
6  end
7  $D_{a\leftarrow}[a] \leftarrow 0$; $D_{b\to}[b] \leftarrow 0$;                *
8  Tag($a$); Tag($b$)
9  while $\exists u \in V$ : COUNT$[u] = \max\{$COUNT$[v] \mid v \in$
   $V\} \wedge \neg$TAGGED$[u]$ do
10     Tag($u$)
11 end
12 return CONSISTENT
```

Instances of this problem represented as an undirected graph are called Simple Temporal Networks (STNs). In an STN $\mathcal{S} = \langle V, E\rangle$, each variable x_i is represented by a vertex $i \in V$, and each constraint $c_{i\leftrightarrow j}$ is represented by an edge $\{i,j\} \in E$ between vertex i and vertex j with two associated weights, viz. $w_{i\to j}$ and $w_{j\to i}$. *Solving* an STN $\mathcal{S}$ is often equated with determining an equivalent minimal network $\mathcal{M}$ (or finding in the process that $\mathcal{S}$ is inconsistent). Such a minimal network has a set of solutions (assignment of values to the variables that is consistent with all constraints) identical to that of the original STN $\mathcal{S}$; however, $\mathcal{M}$ is *decomposable*, which means that any solution can be extracted from it in a backtrack-free manner. For STNs, the minimal network can be determined by enforcing *path consistency* (PC), which in turn coincides with calculating all-pairs shortest paths (APSP) on the constraint graph. Determining APSP for $\mathcal{S} = \langle V, E\rangle$, having n vertices and m edges, yields a complete graph; it takes worst-case time $\mathcal{O}\left(n^3\right)$ or $\mathcal{O}\left(n^2 \log n + nm\right)$ using Floyd–Warshall or Johnson's algorithm, respectively. Note that we always assume the (constraint) graph to be connected, so $m \geq n - 1$.

Instead of enforcing PC on an STN $\mathcal{S}$, one can opt to enforce *partial path consistency* (PPC) (Bliek and Sam-Haroud 1999). Although this yields a network $\mathcal{M}^*$ which is not minimal, $\mathcal{M}^*$ shares with $\mathcal{M}$ the properties of decomposability and equivalence to $\mathcal{S}$. Moreover, $\mathcal{M}^*$ can be used to solve the queries listed in the previous section for those pairs of time points in which one is interested. Furthermore, while $\mathcal{M}$ is represented by a complete graph (having $\Theta\left(n^2\right)$ edges), $\mathcal{M}^*$ is represented by a *chordal graph* (sometimes also called triangulated), requiring that every cycle of length four or more has an edge joining two vertices that are not adjacent in the cycle. Such a chordal graph has $\mathcal{O}\left(nw_d^*\right)$ edges and thus is potentially much sparser than $\mathcal{M}$, where w_d^* is the *graph width* induced by an ordering d of the vertices in V. As in $\mathcal{M}$, all edges $\{i,j\}$ in $\mathcal{M}^*$ are labelled by the lengths $w_{i\to j}$ and $w_{j\to i}$ of the shortest paths from i to j and from j to i, respectively.

Partial path consistency can be enforced by the P^{3}C algorithm in $\mathcal{O}\left(n(w_d^*)^2\right)$ time (Planken, de Weerdt, and van der Krogt 2008). Regarded as the current state of the art for solving an STN non-incrementally, P^{3}C builds on the concept introduced by Xu and Choueiry (2003). (We postpone discussion of existing methods for incrementally solving an STN to a later section.) The minimal possible value of w_d^* is exactly the treewidth of G (denoted simply as w^*).

Determining treewidth is NP-hard in general (Arnborg, Corneil, and Proskurowski 1987). However, if G is already chordal, we can—in $\mathcal{O}(m)$ time, using *maximal cardinality search* (MCS) (West 1996)—find an optimal vertex ordering d, yielding minimal induced width w_d^*. Moreover, even a suboptimal choice of d usually results in a shorter run time than that of PC enforcing algorithms, as empirically demonstrated by Planken, de Weerdt, and van der Krogt (2008).

3. The IPPC Algorithm

Mixed-initiative planning and multi-agent coordination are just two of the applications in which new constraints—that subsume existing constraints—are incrementally imposed, and for which we are interested in maintaining decomposability of an STN representation, be it PC or PPC. Although this can of course be done by repeatedly applying standard single-shot algorithms, a more efficient approach is to use tailored incremental methods.

In this paper, we refer to the two best known algorithms for incrementally maintaining PC as IFPC and IAPSP. IFPC, achieving incremental full path consistency, was first used by Tsamardinos and Pollack (2003) and later explicitly presented by Planken (2008). IAPSP is presented as an incremental method for maintaining APSP by Even and Gazit (1985). This algorithm requires $\mathcal{O}\left(n^2\right)$ space, and $\mathcal{O}\left(n^2\right)$ and $\mathcal{O}\left(m^*\delta\right)$ time, where m^* equals the number of edges in $\mathcal{M}$ whose label is to be updated because of the addition of the new constraint, and δ is the graph degree of the original constraint graph $\mathcal{S}$.

We present an algorithm, called IPPC, that maintains PPC under edge tightening, i.e., when the weight of edges is reduced, in $\mathcal{O}\left(m_c\right)$ and $\mathcal{O}\left(n_c^*\delta_c\right)$ time, where m_c, n_c^* and δ_c are the number of edges in the chordal graph, the number of endpoints of updated edges in the chordal graph and the chordal graph's degree, respectively. The first time bound of IPPC is always better than that of IAPSP's first bound. In general, comparison between the second bounds does not yield a clear winner; however, in graphs with small treewidth IPPC gains the upper hand since δ_c is then not significantly higher than δ, whereas n_c^* is always smaller than m^* regardless of the problem instance. Regarding the space complexity, the algorithm we present requires only $\mathcal{O}\left(m_c\right)$ space, instead of $\mathcal{O}\left(n^2\right)$ for IAPSP. These improved bounds can be obtained because not all shortest paths are maintained, but only shortest partial paths in the chordal graph. As noted, this information is still sufficient to check consistency, and the maintained graph is still decomposable.

Recall that the maximal cardinality search (MCS) algo-

Procedure Tag($v \in V$)

```
1  TAGGED[$v$] ← TRUE
2  forall $u \in \mathcal{N}(v)$ do
3      if ¬TAGGED[$u$] then
4          $D_{a\leftarrow}[u] \leftarrow \min\{D_{a\leftarrow}[u], w_{u\to v} + D_{a\leftarrow}[v]\}$;   *
5          $D_{b\to}[u] \leftarrow \min\{D_{b\to}[u], D_{b\to}[v] + w_{v\to u}\}$;   *
6          COUNT[$u$] ← COUNT[$u$] + 1
7      else
8          $w_{u\to v} \leftarrow$
           $\min\{w_{u\to v}, D_{a\leftarrow}[u] + w'_{a\to b} + D_{b\to}[v]\}$;   *
9          $w_{v\to u} \leftarrow$
           $\min\{w_{v\to u}, D_{a\leftarrow}[v] + w'_{a\to b} + D_{b\to}[u]\}$;   *
10     end
11 end
```

rithm can be used to find an ordering d of the vertices in a chordal graph, yielding minimal induced width $w_d^* = w^*$. This algorithm is at the heart of our incremental approach. The order in which our algorithm visits vertices is determined in exactly the same way; the difference lies in the determination of the new shortest paths resulting from tightening some edge weight $w_{a\to b}$ to a lower value $w'_{a\to b}$.

Pseudo-code for our method IPPC is given as Algorithm 1; if the lines marked with an asterisk at the right-hand margin are left out, we obtain the MCS algorithm. IPPC takes as input a PPC STN and a new or tightened constraint with weight $w'_{a\to b}$ for some $\{a, b\} \in E$. If this weight is smaller than the existing weight $w_{a\to b}$, some other constraints may need to be tightened as well. That is, for some other pairs of vertices $\{u, v\} \in E$, the shortest path between them may be reduced by using the (tightened) edge between a and b. In the course of the algorithm, we therefore compute for every vertex v the length of the shortest path to a, as well as the length of the shortest path from b, which are maintained in arrays $D_{a\leftarrow}[v]$ and $D_{b\to}[v]$, respectively. Note that exactly these operations involving edge weights and distance arrays are additions to the MCS algorithm.

In the procedure Tag(v), we update $w_{u\to v}$ for neighbours u of v, denoted by $\mathcal{N}(v)$, that have already been visited if there is a shorter path from u via a and b to v (and similarly update $w_{v\to u}$ if applicable). In order to achieve this, every vertex v is tagged exactly once; in particular, $w_{a\to b}$ itself is updated when calling Tag(b). The arrays COUNT[] and TAGGED[] are used to ensure that the vertices are visited in a *simplicial construction ordering*. Such an ordering, whose existence is a defining property of chordal graphs, was also employed by the state-of-the-art P^{3}C algorithm and is found by MCS in $\mathcal{O}(m_c)$ time. Addition of the marked lines does not cause this bound to be overstepped.

Lemma 1. *The order in which Algorithm 1 calls* Tag(v) *on vertices $v \in V$ is a simplicial construction ordering of $\mathcal{S}$.*

Proof. The order is solely determined by the arrays COUNT[] and TAGGED[]; IPPC uses them in exactly the same way as MCS does. □

Lemma 2. *Upon entering* Tag(v)*, for $v \neq a$, $D_{a\leftarrow}[v]$ and $D_{b\to}[v]$ are equal to the lengths of the shortest paths from v to a and from b to v, respectively.*

Proof. $D_{a\leftarrow}[v]$ and $D_{b\to}[v]$ can never be lower than the length of a shortest path, because they both start at ∞ for every vertex in line 4 of IPPC and are only reduced (in line 4–5 in Tag(v)) when there is a path from v to a or from b to v. We omit the other case for brevity; it can be found in the online version of this paper at the authors' web sites. □

Theorem 3. *Algorithm 1 correctly re-enforces PPC (or decides inconsistency) in $\mathcal{O}(m_c)$ time and space.*

Proof. If some arc $u \to v$ must be updated due to tightening $a \to b$, its new weight is the length of the path $u \dashrightarrow a \to b \dashrightarrow v$, where $\dashrightarrow$ denotes a shortest path. W.l.o.g. assume that u is tagged before v. Then, when calling Tag(v), by Lemma 2, $D_{a\leftarrow}[u]$ and $D_{b\to}[v]$ are correctly set; in line 8 of Tag(v), $w_{u\to v}$ is correctly updated.

Regarding the run time, note that Tag() is called at most once per vertex, exactly as MCS does. All operations in Tag() require amortised constant time per edge; the run time can thus be bounded by $\mathcal{O}(m_c)$.

In the course of the algorithm, four arrays of length $\mathcal{O}(n)$ are maintained ($D_{a\leftarrow}[]$, $D_{b\to}[]$, COUNT[], TAGGED[]), and only one adjacency-list-based graph data structure of size $\mathcal{O}(m_c)$, containing all weights $w_{u\to v}$. Space can thus be bounded by $\mathcal{O}(m_c)$. □

Improving Efficiency

The efficiency of our new algorithm can be improved by considering only vertices for which the weight of one of the incident edges is to be updated. Consider therefore the induced subgraph $G[V^*]$ of G, where $V^* \subseteq V$ consists of all endpoints of edges whose weight is to be updated after tightening $\{a, b\} \in E$. Note that the (implicit) determination of V^* is part of our algorithm. We show that all updated shortest paths can be traced in $G[V^*]$, and that we therefore only need to consider vertices once we have updated one of their incident edges in lines 8–9 of Tag(v).

Lemma 4. *For all $v, w \in V^*$, any updated shortest path $v \dashrightarrow w$ can be traced in $G[V^*]$.*

Proof. Assume the contrary, i.e. $\exists v, w \in V^*$ for which $w_{v\to w}$ needs to be updated, but $v \dashrightarrow w$ cannot be traced in $G[V^*]$. (In case there are multiple shortest paths $v \dashrightarrow w$, consider any one of them with the fewest number of edges.) There is a vertex $u \notin V^*$ on this shortest path $v \dashrightarrow a$ or on $b \dashrightarrow w$. We give proof in case u is on $v \dashrightarrow a$; the other case is analogous. There is a cycle $v \dashrightarrow u \dashrightarrow a \to b \dashrightarrow w \to v$ in G, and since the path from v to a has the fewest number of edges, there is no chord from v to any vertex on $u \dashrightarrow a$. Thus, and since G is chordal, there must be a chord incident on u. For the same reason, this chord cannot connect u to any vertex on $u \dashrightarrow a$. Thus, the chord must be between u and a vertex u' on $b \dashrightarrow w$. Since $w_{v\to w}$ needs an update, $v \dashrightarrow u \dashrightarrow a \to b \dashrightarrow u' \dashrightarrow w$ is shorter than $v \dashrightarrow u \to u' \dashrightarrow w$. Therefore, the path $u \dashrightarrow a \to b \dashrightarrow u'$ is shorter than $u \to u'$, and thus $w_{u\to u'}$ needs to be updated. But then $u \in V^*$, a contradiction. □

This lemma suggests a change the algorithm to ensure that exactly all vertices in V^* are completely visited.

Procedure Tag–improved($v \in V$)

```
1  TAGGED[$v$] ← TRUE
2  forall $u \in$ LIST[$v$] do
3      $w_{u \to v} \leftarrow \min\{w_{u \to v}, D_{a\leftarrow}[u] + w'_{a \to b} + D_{b\to}[v]\}$
4      $w_{v \to u} \leftarrow \min\{w_{v \to u}, D_{a\leftarrow}[v] + w'_{a \to b} + D_{b\to}[u]\}$
5  end
6  if any updates took place, or $v = a$ then
7      forall $u \in \mathcal{N}(v)$ such that ¬TAGGED[$u$] do
8          $D_{a\leftarrow}[u] \leftarrow \min\{D_{a\leftarrow}[u], w_{u \to v} + D_{a\leftarrow}[v]\}$
9          $D_{b\to}[u] \leftarrow \min\{D_{b\to}[u], D_{b\to}[v] + w_{v \to u}\}$
10         LIST[$u$] ← LIST[$u$] ∪ {$v$}
11     end
12 end
```

We therefore present a new procedure Tag–improved() which replaces the original Tag(). The array COUNT[] is replaced by an array of vertex lists LIST[], initialised to contain an empty list for each vertex. These lists contain for each vertex the neighbouring vertices in V^* that have already been tagged; in this manner, the algorithm only needs to iterate over all of the vertex's neighbours if the vertex itself is found to be in V^* because an update took place (or the vertex in question is a which is trivially in V^*). In the main algorithm, instead of checking vertices' counts, the length of these lists is checked; and Tag–improved() is only called when this length is at least two (it is easily verified that otherwise, the vertex does not appear in any cycle with the updated edge, so no checking is necessary). Recall that n_c^* and δ_c are the number of endpoints of updated edges in the chordal graph and its degree, respectively.

Theorem 5. *With the modifications discussed above, algorithm 1 correctly re-enforces PPC (or decides inconsistency) in $\mathcal{O}(m_c)$ and $\mathcal{O}(n_c^* \delta_c)$ time.*

Proof. The modifications ensure that exactly all n_c^* vertices in V^* are completely visited. Edges incident on vertices outside V^* do not need to be updated to maintain partial path consistency. From Lemma 4 we can then conclude that updates on edges incident on vertices from V^* can correctly be done when restricting the algorithm to $G[V^*]$. Regarding the run time, the first bound is proven in Theorem 3 and still applies, because the inserted lines all take constant amortised time per edge. The second bound can be derived as follows. Only vertices in V^* are completely visited, and when they are, all neighbours in the chordal graph are visited. Each vertex has at most δ_c neighbours in the chordal graph, and each visit takes only constant time (amortised) per edge. Consequently, the run time can be bounded by $\mathcal{O}(n_c^* \delta_c)$. □

4. Existing Algorithms

We briefly review existing methods for incrementally solving an STN. The list of methods we include here is not complete; especially in the case of incremental algorithms, many other approaches exist, some of which are designed for special cases, e.g. where all edge weights are positive. We give particular attention to incremental algorithms that have been used as components of solvers for the Disjunctive Temporal Problem (DTP) (Stergiou and Koubarakis 2000), as the DTP has spurred advances in incremental STN solving.

The Bellman–Ford algorithm can be used as a single-shot approach if one is interested only in determining consistency, or single-source shortest paths. Cesta and Oddi (1996) published an incremental variant tailored to the STN that retains a theoretical upper bound on the run time of $\mathcal{O}(mn)$. Further work on incremental consistency checking for the STN has been done by Ramalingam et al. (1999), who also list some algorithms for special cases of the STN.

It is also possible to incrementally maintain full path consistency (IFPC) explicitly. Tsamardinos and Pollack (2003) use this approach, citing Mohr and Henderson (1986); however, instead of an incremental approach, the latter authors presented (only) a new single-shot path consistency algorithm for general constraint satisfaction problems. Nevertheless, an incremental version can be obtained that operates within a time bound of $\mathcal{O}\left(n + n^{*2}\right)$, as shown by Planken (2008)—although the time bound was left implicit. We note that n^* is never lower than n_c^*, and may be much higher; e.g. in a tree, $n_c^* = 2$ while n^* may be as high as n. Recall that for STNs, this approach is equivalent to calculating APSP.

Demetrescu and Italiano (2006a) give an overview of available algorithms for the *dynamic* all-pairs shortest paths problem. They also consider the problem with only positive edge-weights, and moreover list algorithms that can handle not only decreasing edge weights and edge additions, but also increases in edge weights and edge deletions; hence the name 'dynamic APSP' instead of 'incremental APSP'. The same authors (Demetrescu and Italiano 2006b) compare empirically a number of algorithms.

Thorup (2004) presents an algorithm for dynamic APSP that allows negative edge weights, based on graph-theoretic combinatorial properties. This algorithm requires $\mathcal{O}\left(n^2(logn + log^2((m+n)/n))\right)$ amortized time per update, and $\mathcal{O}(mn)$ space.

Hunsberger (2008) presents a complete temporal constraint management system to support planning, scheduling, and real-time execution, based on an STN in which so-called *rigid components* are collapsed (in the case of constraint tightening only). These rigid components are subgraphs where for every edge one weight is equal to the negative of the other. Attention is paid to practical space complexity as well as time complexity. The temporal network separates the temporal reference time point into a pair and includes a 'now' time point; together, these eliminate the need for certain common forms of constraint propagation when considering the passage of time. The constraint propagation algorithm is based on a two-phase pass over undominated paths propagated from a changed edge. It provides dynamic APSP information; a variant can provide incremental APSP information with complexity $O(m^*\delta)$. This algorithm is essentially the same as the one introduced by Even and Gazit (1985), which we dubbed IAPSP, and can be shown to have the best attainable general upper bound on the time complexity for an incremental APSP algorithm. The later improvements by Hunsberger (2008) do not change the theoretical upper bound, but improve efficiency in practice. These improvements can be applied to IPPC as well. However, in the

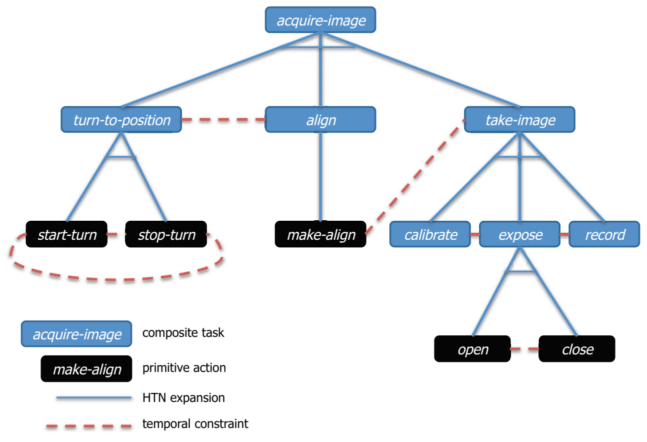

Figure 1: An example of an HTN in a satellite domain.

next section we compare the performance of IAPSP to that of IPPC, both without these improvements.

One earlier attempt to define an incremental algorithm based on PPC has been made (Planken 2008). The approach taken is, roughly, to perform the second half of the P^{3}C algorithm, with a few improvements. The time complexity is $\mathcal{O}\left(n(w_d^*)^2\right)$, i.e. equal to that of P^{3}C, which is worse than the other methods in the literature, and the empirical performance was likewise disappointing. Our new approach for an incremental PPC algorithm is completely different; it makes more use of the properties of chordal graphs and enjoys tighter theoretical bounds.

5. Experimental Evaluation

We next empirically compare our new IPPC algorithm to the state of the art in incremental path consistency, i.e. the IAPSP algorithm by Even and Gazit (1985), and also IFPC as described by Planken (2008).

We evaluate the performance of IPPC on two sets of benchmarks, which we believe are representative of the temporal sub-problems encountered when solving planning or scheduling problems. The first set of STNs are extracted from a set of job-shop scheduling benchmark problems. The second set of STNs are so-called sibling-restricted STNs, which are obtained from (automatically generated) Hierarchical Task Networks (HTNs).

Job-shop STNs

Job-shop problem instances of different sizes are included in SMT-LIB (Ranise and Tinelli 2003). The temporal sub-problem in a job-shop problem consists of checking consistency of a given ordering of jobs. We thus generate STN sub-problems by creating constraints based on a guessed order of the jobs, and then change the weights of the STN to ensure their consistency while maintaining structure. Because of the fixed size of the benchmark problems in SMT-LIB, this process results in STNs with up to 250 vertices.

Sibling-Restricted STNs

The Hierarchical Task Network (HTN) planning paradigm (Erol, Hendler, and Nau 1994) assumes a hierarchical flow, with high-level tasks being decomposed progressively into collections of lower-level tasks through the application of matching methods with satisfied preconditions. The HTN planning process gives rise to STNs with the *sibling-restricted* (SR) property (Bui, Tyson, and Yorke-Smith 2008). In a sibling-restricted STN, constraints may occur only between parent tasks and their children, and between sibling tasks. This restriction on what STN constraints may exist between plan elements is inherent to HTN planning models; in particular, there is no way in standard HTN representations to specify temporal constraints between tasks in different task networks (Erol, Hendler, and Nau 1994). Hence, standard HTN representations have been extended to support limited coordination between different task networks via *landmark variables* (Castillo, Fdez-Olivares, and O. Garca-Pérez 2006) that allow synchronisation of key events in the plan.

Apart from the temporal reference time point TR, vertices in the STN derived from an HTN occur in pairs, because each task in the HTN corresponds to two variables, the start and end time points. It follows that the constraint graph is biconnected and decomposes only via separator sets of cardinality at least two. The structure in HTN-derived STNs is exploited by single-shot temporal solvers such as Prop-STP (Bui, Tyson, and Yorke-Smith 2008).

An HTN planning scenario in a satellite domain is given by Figure 1; Figure 2 depicts an STN that arises. The top-level task, *acquire-image*, is being achieved by a task network with three second-level tasks, *turn-to-position*, *align*, and *take-image*. In the HTN, selected temporal constraints are shown; implied temporal constraints from the HTN structure are not shown. In the STN, all temporal constraints are shown, except unary constraints from the temporal reference time point TR to other time points. An omitted edge weight corresponds to the unconstraining weight $[0, \infty)$. Interval notation is used: the lower bound of an interval corresponds to a reverse arc with negative weight. Thus, the STN has many arcs labelled -10, and one each labelled -1 and -2.

As the HTN planning process proceeds, task networks are considered for matching with non-ground tasks. When each candidate network is considered, its time points and edges, and specified edges between its time points and existing time points, are added to the STN. Thus, incrementally the STN is built up; propagation is required at each step. The propagation determines if the candidate network will cause inconsistency (indicating that an alternate network is required, or if none exist, backtracking). It also narrows the temporal bounds (i.e. the domains) for the time points.

In the scenario, suppose that a matching task network for task *expose* is being considered. If the depicted network, consisting of primitive actions *open* and *close*, and the temporal constraint between them, is added, then the corresponding time point pairs and edges are added to the STN, as shown, and incremental propagation should be performed.

As noted, landmark time points are introduced in order to support limited coordination between different task networks. This allows synchronisation of key events in the plan, but breaks the full sibling-restricted property. Such a landmark time point may be an existing task start or end, or it

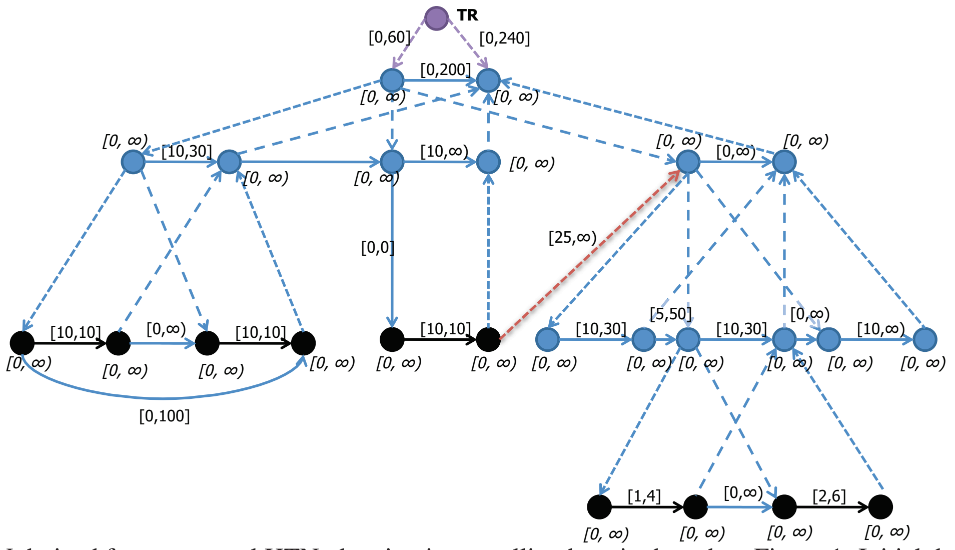

Figure 2: STN derived from temporal HTN planning in a satellite domain, based on Figure 1. Initial domains shown.

may be additional time point that does not equate to the start or end of any task. In Figure 1, *take-image-start* is an example of the former.

We generate HTNs using the following parameters: (i) the branching factor, determining the number of children for each node, (ii) the depth of the HTN tree, (iii) the ratio of landmark time points to the number of tasks in the HTN, and (iv) the ratio of constraints between siblings versus the total number of possible sibling constraints (which is the branching factor squared). From the worst-case bounds, we expect IPPC to perform especially well in sparse graphs, such as when the landmark ratio and sibling ratio are small.

Experimental Setup

The benchmark sets described above contain large STNs. As discussed before, for IPPC to run, the constraint graph must be chordal. In our experiments we therefore always first construct the graph consisting of all constraint edges that will be added and triangulate it once with the minimum-fill heuristic. We thus reserve incrementally maintaining chordality for future work. It is known that this can be done time at most linear in the number of vertices (Berry, Heggernes, and Villanger 2006; Ibarra 2008).

To evaluate the performance of incrementally solving these STNs, we start with an empty graph and add new constraints with weights $w'_{a\to b}$ one by one. We run our implementation of the algorithm under evaluation in a Java virtual machine with 1 GB of memory assigned on a 2.4 GHz AMD processor, and measure the accumulated time needed to add all constraints.

Results

The results of the experiment on the job-shop STNs can be found in Figure 3. Surprisingly, the very simple IFPC algorithm clearly outperforms the others on these instances, showing even better performance than reported by its originators (Planken 2008). The run times of IPPC and IAPSP

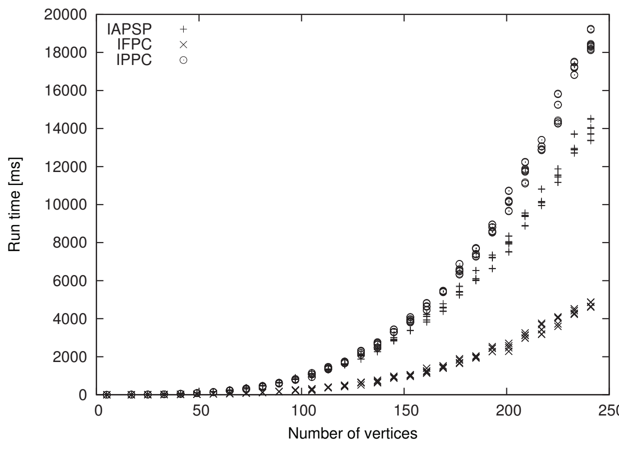

Figure 3: IFPC run time outperforms both IAPSP and IPPC on the job-shop benchmarks.

are quite close, although IAPSP is slightly better. The results are all the more surprising because the theoretical upper bounds for these instances are better for IPPC than for either IFPC or IAPSP, as can be seen in Figure 4; again, it is notable that the very simple approach taken by IFPC has a better theoretical upper bound than the more complex IAPSP algorithm. We believe that these differences can be attributed to the fact that the bounds on IAPSP and IFPC are not as tight as the bound on IPPC, and, further, some of the ideas in the implementation of IAPSP such as making efficient use of a shortest path tree have not (yet) been used to improve IPPC.

To model a representative set of HTN benchmarks we vary the branching factor between 4 and 10, the depth of each branch between 3 and 7, set the landmark ratio to 0.15, and the sibling ratio to 0.5. The results for these HTN benchmarks show a completely different picture than the job-shop benchmarks, as can be seen in Figure 5. For large problem instances, the run times of IAPSP and IFPC are significantly

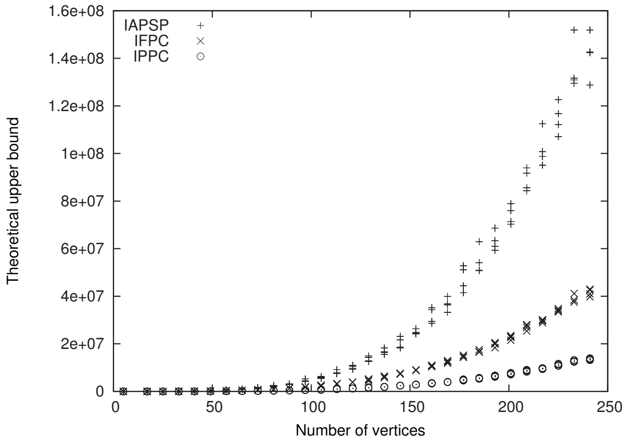

Figure 4: For the job-shop benchmarks, the theoretical upper bound on the run time of IPPC is lower than those for both IAPSP and IFPC.

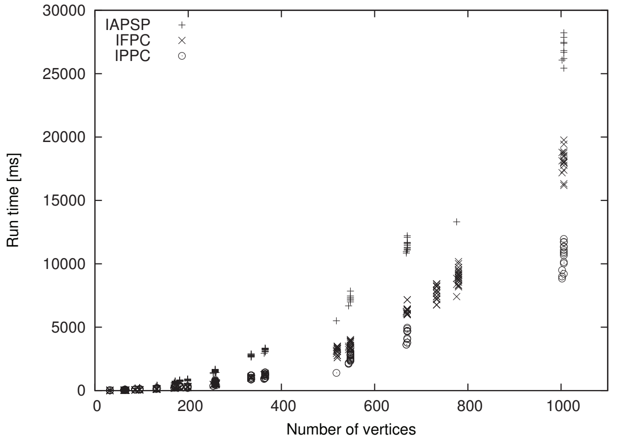

Figure 5: IPPC outperforms IAPSP and IFPC on the HTN benchmarks.

higher than that of IPPC.

In a similar experiment, we studied the influence of the number of sibling constraints, varying the ratio of sibling constraints to possible sibling constraints between 0.1 and 0.9, both for a setting with landmark ratio 0.0 as well as with landmark ratio 0.2. The result of this experiment is surprising. Increasing the sibling ratio seems to have no effect on the performance of the algorithms at all. (The resulting graphs are therefore uninteresting, and are omitted here due to space limitations.) We believe this result may occur as, although these constraints increase the density, they do so only in a very local manner, having only a marginal influence on the total number of weights to update.

In a final experiment, we varied the landmark ratio from 0.0 to 0.3. The effect of increasing the density in this manner is the expected behaviour, shown in Figure 6. IPPC is slower on problem instances with a high landmark ratio, but for sparse instances outperforms the other algorithms.

From these experiments we conclude that IPPC can be competitive with IAPSP when problem instances are derived from for example job-shop scheduling, but that IFPC is a surprisingly serious alternative. However, when dealing with STNs that occur as a sub-problem of HTNs with low enough landmark ratio, IPPC is the most efficient method.

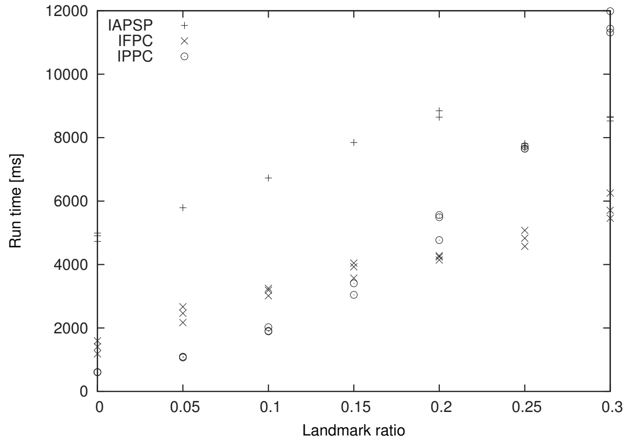

Figure 6: IPPC outperforms IAPSP and IFPC when the ratio of the landmarks versus other vertices is limited.

6. Conclusions and Future Work

When solving temporal planning or scheduling problems, consistency of temporal constraints is frequently checked, and temporal propagation is frequently performed. Prior to the work reported here, the most successful approach has been to incrementally maintain all-pairs-shortest-paths (IAPSP) (Even and Gazit 1985; Ramalingam et al. 1999; Hunsberger 2008), or full path consistency (IFPC) (Tsamardinos and Pollack 2003; Planken 2008). The primary contribution of this paper is to demonstrate that these algorithms perform unnecessary work, especially in sparse graphs. We leverage recent results that have shown that enforcing partial path consistency (Xu and Choueiry 2003; Planken, de Weerdt, and van der Krogt 2008) is sufficient to answer the relevant temporal inference questions, and we propose the first successful method to enforce partial path consistency incrementally.

Our algorithm, IPPC, has a worst-case time complexity bounded by the number of edges in the chordal graph, which is better than the bound on IAPSP of the number of vertices squared. The time complexity can also be bounded by the degree of the chordal graph times the number of endpoints of updated edges in the chordal graph. For sparse graphs, this latter bound is also better than that of IAPSP, which is the degree times the number of updated edges in the original graph, as well as better than that of IFPC, which is the number of vertices plus the number of updated vertices squared. In addition, IPPC requires space only linear in the number of edges of the chordal graph, whereas earlier work uses space quadratic in the number of vertices. We also find that IFPC's bound is lower than that of the other algorithms on job-shop STNs, a surprising result given its simplicity.

In the experiments presented in this paper, we compared IPPC to an efficient implementation of IAPSP as described by Even and Gazit (1985), and an implementation of IFPC described by Planken (2008). Ausiello et al. (1991) describe an algorithm which can be adapted for the incremental APSP problem, that generally shows better empirical performance than IFPC; more recently, Hunsberger (2008) has proposed some ideas for IAPSP which do not improve the worst-case bound, but can reduce the computational load in certain cases (e.g., splitting the reference point and collapsing rigid components). As part of our future work we plan to include these ideas both in our implementation of IAPSP as well as in our implementation of IPPC, and then compare these improved methods experimentally.

Secondly, in this paper we focused on enforcing partial path consistency incrementally, assuming the constraint graph is already chordal. For the method to be fully incremental, it should also allow addition of new edges which may break chordality. To this end we plan on including one of the recently discovered efficient algorithms for maintaining chordality (Berry, Heggernes, and Villanger 2006; Ibarra 2008). Since these methods take time at most linear in the number of vertices, we do not expect their incorporation to result in any significant change in performance. Thirdly, we plan to develop methods to also deal with constraint loosening and edge deletions. With these additions, we expect to arrive at a fully dynamical method that not only performs significantly better than any existing approach for dynamically solving STNs, but can immediately be used as part of a planning or scheduling algorithm.

References

Arnborg, S.; Corneil, D. G.; and Proskurowski, A. 1987. Complexity of finding embeddings in a k -tree. *SIAM Journal on Algebraic and Discrete Methods* 8(2):277–284.

Ausiello, G.; Italiano, G.; Marchetti-Spaccamela, A.; and Nanni, U. 1991. Incremental algorithms for minimal length paths. *J. Algorithms* 12(4):615–638.

Berry, A.; Heggernes, P.; and Villanger, Y. 2006. A vertex incremental approach for maintaining chordality. *Discrete Mathematics* 306(3):318–336.

Bliek, C., and Sam-Haroud, D. 1999. Path consistency on triangulated constraint graphs. In *Proc. of IJCAI-99*, 456–461.

Bresina, J.; Jónsson, A. K.; Morris, P.; and Rajan, K. 2005. Activity planning for the Mars exploration rovers. In *Proc. of ICAPS-05*, 40–49.

Bui, H. H.; Tyson, M.; and Yorke-Smith, N. 2008. Efficient message passing and propagation of simple temporal constraints: Results on semi-structured networks. In *Proc. of CP/ICAPS'08 Joint Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, 17–24.

Castillo, L.; Fdez-Olivares, J.; and O. Garca-Pérez, F. P. 2006. Efficiently handling temporal knowledge in an HTN planner. In *Proc. of ICAPS-06*, 63–72.

Cesta, A., and Oddi, A. 1996. Gaining efficiency and flexibility in the simple temporal problem. In *Proc. of TIME'96*, 45–50.

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

Demetrescu, C., and Italiano, G. 2006a. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms* 4(3):353–383.

Demetrescu, C., and Italiano, G. 2006b. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Transactions on Algorithms* 2(4):578–601.

Erol, K.; Hendler, J.; and Nau, D. 1994. Semantics for Hierarchical Task-Network planning. Technical Report CS-TR-3239, University of Maryland.

Even, S., and Gazit, H. 1985. Updating distances in dynamic graphs. *Methods of Operations Research* 49:371–387.

Hunsberger, L. 2008. A practical temporal constraint management system for real-time applications. In *Proc. of ECAI-08*, 553–557.

Ibarra, L. 2008. Fully dynamic algorithms for chordal graphs and split graphs. *ACM Transactions on Algorithms* 4(4):40:1–20.

Laborie, P., and Ghallab, M. 1995. Planning with sharable resource constraints. In *Proc. of IJCAI-95*, 1643–1649.

Mohr, R., and Henderson, T. C. 1986. Arc and path consistency revisited. *Artificial Intelligence* 28(2):225–233.

Planken, L. R.; de Weerdt, M. M.; and van der Krogt, R. P. 2008. P3C: A new algorithm for the simple temporal problem. In *Proc. of ICAPS-08*, 256–263.

Planken, L. R. 2008. Incrementally solving the STP by enforcing partial path consistency. In *Proc. of 27th PlanSIG Workshop*, 87–94.

Ramalingam, G.; Song, J.; Joskowicz, L.; and Miller, R. E. 1999. Solving systems of difference constraints incrementally. *Algorithmica* 23(3):261–275.

Ranise, S., and Tinelli, C. 2003. The SMT-LIB format: An initial proposal. In *Proc. of PDPAR'03*, 94–111.

Smith, D. E.; Frank, J.; and Jónsson, A. K. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review* 15(1):47–83.

Stergiou, K., and Koubarakis, M. 2000. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence* 120(1):81–117.

Thorup, M. 2004. Fully-dynamic all-pairs shortest paths: Faster and allowing negative cycles. In *Proc. of 9th Scandinavian Workshop on Algorithm Theory*, 384–396.

Tsamardinos, I., and Pollack, M. E. 2003. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence* 151(1–2):43–89.

West, D. B. 1996. *Introduction to Graph Theory*. Prentice-Hall.

Xu, L., and Choueiry, B. Y. 2003. A new efficient algorithm for solving the Simple Temporal Problem. In *Proc. of TIME-ICTL-03*, 210–220.