Maintaining Partial Path Consistency in STNs under Event-Incremental Updates

Ot ten Thije and Léon Planken and Mathijs de Weerdt
Delft University of Technology

Abstract

Efficient management of temporal constraints is important for temporal planning. During plan development, many solvers employ a heuristic-driven backtracking approach, over the course of which they maintain a so-called Simple Temporal Network (STN) of events and constraints. This paper presents the Vertex-IPPC algorithm, which efficiently enforces partial path consistency when an STN is extended with an event and all its associated constraints. This algorithm uses some new results on directed path consistency on subgraphs. We prove that the worst-case time complexity of our algorithm is competitive with extant approaches. Our algorithm integrates well with a recently discovered vertex-incremental triangulation method, which, to the best of our knowledge, we are the first to have implemented. While experiments show that the new algorithm is outperformed on realistic networks, it is competitive on chordal graphs.

1 Introduction

Quantitative temporal constraints are essential for many real-life planning and scheduling domains (Smith et al., 2000). The Simple Temporal Network (STN; Dechter et al., 1991) is a popular model for temporal events and simple temporal constraints among them. The central role of STNs in deployed planning systems (Bresina et al., 2005; Castillo et al., 2006; Labone and Ghallab, 1995) 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 (Planken, 2008; Xu and Choueiry, 2003), answering queries such as:

• Is the information represented by the STN consistent?
• How can the events be scheduled to meet all constraints?
• What are possible times at which event $x_i$ could occur?
• What relation between two events $x_i$ and $x_j$ is implied?

Dechter et al. (1991) identified these types of queries as the most relevant when solving STNs.

This paper introduces a new algorithm dubbed Vertex-IPPC, which maintains partial path consistency when a set of constraints associated with a new event is added to an STN. Vertex-IPPC is complementary to the IPPC algorithm designed by Planken et al. (2010), which maintains partial path consistency when a new constraint is added. To emphasise the difference between the two approaches, we will refer to the original IPPC as Edge-IPPC from here onward.

We first give the necessary definitions and discuss related techniques. Then we show that the concept of directed path consistency also applies to subgraphs, and how this can be used in our algorithm. We present proofs of correctness and upper bounds on run time. We then empirically validate the derived bounds of Vertex-IPPC and compare it with a straightforward extension of Edge-IPPC. 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 $c_{i \rightarrow j}$ has a weight $w_{i \rightarrow j} \in \mathbb{R} \cup \{\infty\}$ corresponding to an upper bound on the difference, and thus represents an inequality $x_j - x_i \leq w_{i \rightarrow j}$. Two constraints $c_{i \rightarrow j}$ and $c_{j \rightarrow i}$ can be combined into a single constraint $c_{i \rightarrow j} : -w_{i \rightarrow j} \leq x_j - x_i \leq w_{i \rightarrow j}$ or, equivalently, $x_j - x_i \in [-w_{i \rightarrow j}, w_{i \rightarrow j}]$, giving both upper and lower bounds. An unspecified constraint is equivalent to a constraint with an infinite weight; therefore, if $c_{i \rightarrow j}$ exists and $c_{j \rightarrow i}$ does not, we have $c_{i \rightarrow j} : x_j - x_i \in (-\infty, w_{i \rightarrow j}]$.

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 $v_i \in V$, and each constraint $c_{i \rightarrow j}$ is represented by an edge $\{i, j\} \in E$ between vertex $v_i$ and vertex $v_j$ with two associated weights, viz. $w_{i \rightarrow j}$ and $w_{j \rightarrow i}$. 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, $\mathcal{M}$ has the property 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. Determin-
Since our algorithm is closely related to the P3C algorithm, having \( n \) vertices and \( m \) edges, yields a complete graph; it takes worst-case time \( \mathcal{O}(n^3) \) or \( \mathcal{O}(n^2 \log n + mn) \) 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 \( S \), one can opt to enforce partial path consistency (PPC; Bliek and Sam-Haroud, 1999). Although this yields a network \( M^* \) which is not minimal, it is nonetheless equivalent to \( S \) and a solution can still be extracted without backtracking. Moreover, \( M^* \) can be used to answer the queries listed in the introduction for those pairs of time points in which one is interested. Furthermore, while \( M \) is represented by a complete graph (having \( \Theta(n^2) \) edges), \( M^* \) 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. We denote the number of edges in such a chordal graph by \( m_c \). Since \( m_c = \mathcal{O}(nw_d) \), where \( w_d \) is the graph width induced by an ordering \( d \) of the vertices in \( V \), \( M^* \) is potentially much sparser than \( M \). As in \( M \), all edges \( \{i,j\} \) in \( M^* \) are labelled by the lengths \( w_{i\rightarrow j} \) and \( w_{j\rightarrow i} \) of the shortest paths from \( i \) to \( j \) and from \( j \) to \( i \), respectively. In summary, an STN is partially path consistent if it is (i) chordal and (ii) every edge is labelled by the weights of the shortest paths between its endpoints.

Partial path consistency can be enforced by the P3C algorithm in \( \mathcal{O}(n(w_d)^2) \) time (Planken et al., 2008). Regarded as the current state of the art for solving an STN non-incrementally, P3C builds on the concept introduced by Xu and Choueiry (2003). The minimum possible value of \( w_d \) is exactly the treewidth of the graph.

Determining treewidth is NP-hard in general (Arnborg et al., 1987). However, if the graph is already chordal, we can—in \( \mathcal{O}(m) \) time, using lexicographic breadth-first search (Lex-BFS: West et al., 2001)—find a so-called simplicial elimination ordering \( d \), a vertex ordering yielding minimum 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 et al. (2008).

Simplicial elimination orderings also disclose a useful structural property of chordal graphs. In particular, every vertex \( v_k \) in a simplicial elimination ordering \( d = \langle v_n, v_{n-1}, \ldots, v_1 \rangle \) of the graph \( G \) is simplicial in the graph \( G_k \) induced by the vertices \( \{v_1, \ldots, v_k\} \), i.e. all vertices \( N_k(v_k) = \{v_i \mid \{v_i, v_k\} \in E, i < k\} \) neighbouring on \( v_k \) in \( G_k \) induce a clique.

### The P3C Algorithm

Since our algorithm is closely related to the P3C algorithm, we will provide some more details about it here. P3C requires the network to be chordal and takes a simplicial elimination ordering as input. It then performs two sweeps along this ordering.

The first sweep uses the DPC algorithm (reproduced here as Algorithm 1) to enforce a property called directed path consistency, introduced by Dechter et al. (1991). This property guarantees that every edge \( \{v_i, v_j\} \) is labelled with the weight of the shortest \( v_i \rightarrow v_j \) path through the subgraph induced by all vertices that precede both \( v_i \) and \( v_j \) in the ordering \( d \). Formally, it is defined as follows:

**Definition 1.** A chordal network \( S \) is DPC along the simplicial elimination ordering \( d = \langle v_n, v_{n-1}, \ldots, v_1 \rangle \) (also called d-DPC) if \( w_{i\rightarrow j} \leq w_{i\rightarrow k} + w_{k\rightarrow j} \) for all \( i, j < k \) where \( v_i, v_j \in N(v_k) \).

The second sweep then follows the reverse of the elimination ordering \( d \), using the fact that the STN is already DPC to inductively compute the globally shortest paths required for the graph to be PPC.

### 3 More on DPC

Our new algorithm relies on two key properties concerning DPC. First, if an STN is PPC, we can show that it is DPC along any simplicial elimination ordering. Second, it follows from the DPC algorithm that if the constraints adjacent to the last vertex in the ordering change, they cannot affect the constraints that have already been eliminated. In other words, the only constraints that need to be updated in this case to re-enforce DPC are those in the direct neighbourhood of the last event in the ordering. In this section we prove these two properties.

### PPC and DPC

**Lemma 1.** Given a chordal STN \( S \) which is PPC. Then, \( S \) is DPC along any simplicial elimination ordering.

**Proof.** Since \( S \) is PPC, all edges are labelled with the length of the shortest path. Thus, for all triangles \( i, j, k \) it holds that \( w_{i\rightarrow j} \leq w_{i\rightarrow k} + w_{k\rightarrow j} \). Now, let \( d \) be any simplicial elimination ordering. Since running DPC along \( d \) also does not introduce any new edges and no weights are adjusted, \( S \) is DPC along \( d \). \( \square \)

Next, we discuss how to select an ordering that minimises the amount of updates.
Lexicographic Breadth-First Search

As can be seen in Algorithm 1, when the DPC algorithm visits a vertex $v_k$ in the simplicial elimination ordering $d$, it only considers the constraints between $v_k$ and its direct, lower-numbered neighbours when adjusting weights. The weights of these neighbouring constraints may be affected by changes made earlier in the traversal. Now, if $v_k$ appears in $d$ before a newly inserted vertex $a$, and is not one of its neighbours, we observe that the DPC algorithm will not adjust any weights during its visit of its neighbours. We would therefore like an ordering in which $a$ and its neighbours are visited as late as possible, ideally last. Assuming that the graph is chordal after insertion of vertex $a$, then we observe that the DPC algorithm will re-enforce DPC in the entire graph.

**Lemma 2.** Given a chordal graph $G = (V, E)$ and a vertex $a \in V$ having $\delta(a)$ neighbours, the simplicial elimination ordering $d$ of $G$ produced by Lex-BFS starting at $a$ satisfies $d(n) = a$ and $\{d(n - \delta_i(a)), \ldots, d(n - 1)\} = N(a)$.

Lex-BFS is a breadth-first search procedure where all vertices receive a label upon their first check, and labels are extended with the current value of a global counter. Vertices are then visited in lexicographical order of this label.

**DPC on induced subgraphs**

We now use Lemma 2 to demonstrate that recomputing DPC along this ordering in the subgraph induced by $\{a\} \cup N(a)$ only is sufficient to re-enforce DPC for the entire graph.

This is done in three steps. First we demonstrate that the subsequence $d'$ of a simplicial elimination ordering $d$ is a valid simplicial elimination ordering for the subgraph induced by the vertices in $d'$. We then revisit the definition of DPC, and show that it can also be expressed as a property of vertices relative to an ordering. Finally, we show that running DPC along the ordering in the subgraph induced by $\{a\} \cup N(a)$ re-enforces this property for all vertices, thus re-enforcing DPC on the entire graph.

Let $G \setminus \{v\}$ be a shorthand for $G \setminus \{v\}$, the graph induced by all vertices in $V$ other than $v$. In other words, $G \setminus \{v\}$ is the graph from which the vertex $v$ and all edges connected to $v$ have been removed. We then have the following result:

**Lemma 3.** Let $d = (v_n, v_{n-1}, \ldots, v_1)$ be a simplicial elimination ordering of the chordal graph $G$. Then the ordering $d' = (v_n, \ldots, v_{k+1}, v_k, \ldots, v_1)$ is a simplicial elimination ordering of the graph $G' = G \setminus \{v_k\}$ for every $1 \leq k \leq n$.

**Proof.** First consider the vertices $v_i$ with $i > k$. Since $d$ is a simplicial elimination ordering of $G$, the vertices $C_i = \{v_j \mid v_j \in N(v_i), j < i\}$ induce a clique in $G$. A subset of vertices in a clique also induces a clique, so $C_i \setminus \{v_k\}$ is a clique in $G'$. Therefore, $v_i$ remains simplicial in $d'$.

Furthermore, for $v_i$ with $i < k$, we know that $C_i$ is a clique in $G$ which cannot contain $v_k$ since $i < k$. Since $G$ and $G'$ only differ in the removal of $v_k$, this clique remains unchanged in $G'$, and $v_i$ remains simplicial in $d'$.

Since $v_k$ itself does not occur in $d'$ and all other vertices remain simplicial, $d'$ must indeed be a valid simplicial elimination ordering of $G'$.

Let $d' = d\setminus V'$ be a sequence of all vertices $v \in V \setminus V'$, appearing in the same order as they appear in $d$. Repeatedly applying Lemma 3 for different vertices of the graph $G$ yields the following corollary:

**Corollary 1.** Given a simplicial elimination ordering $d$ of a chordal graph $G = (V, E)$ and an arbitrary subset $V' \subseteq V$. Then the ordering $d'' = d\setminus V'$ is a simplicial elimination ordering for the graph $G' = G\setminus V'$.

Given an elimination ordering for the entire graph $G$, Corollary 1 allows us to derive an elimination ordering for the subgraph $G(a) \cup N(a)$ induced by $\{a\} \cup N(a)$. We now need to show that re-enforcing DPC in this subgraph suffices to re-enforce DPC in the entire graph.

Unfortunately, DPC is defined in terms of an ordering over the entire graph. For Vertex-IPPC, we want to be able to reason about DPC in the context of an induced subgraph. Definition 2 addresses this issue by providing an alternative definition of DPC at the level of individual vertices.

**Definition 2.** A vertex $v_k$ of an STN $S$ has the DPC property relative to the ordering $d = (v_n, v_{n-1}, \ldots, v_1)$ if $w_{i\rightarrow j} \leq w_{i\rightarrow k} + w_{k\rightarrow j}$ for all $i, j < k$ where $v_i, v_j \in N(v_k)$.

The following proposition shows that these definitions express the same notion of directed path consistency.

**Proposition 4.** A network $S$ is DPC along the ordering $d$ (d-DPC) if and only if all its vertices have the DPC property relative to $d$.

**Proof.** ($\Rightarrow$) If $S$ is DPC along $d$, by Definition 1 the inequality $w_{i\rightarrow j} \leq w_{i\rightarrow k} + w_{k\rightarrow j}$ holds for every vertex $v_k$, and hence every $v_k$ has the DPC property relative to $d$.

($\Leftarrow$) For the other direction, if every vertex has the DPC property relative to $d$, we know by Definition 2 that the inequality $w_{i\rightarrow j} \leq w_{i\rightarrow k} + w_{k\rightarrow j}$ holds for every $v_k$, and therefore that $S$ is DPC along $d$.

We use the shorthand d-DPC both to indicate that a vertex has the DPC property relative to $d$ and to indicate that an STN is DPC along $d$.

The concept of the DPC property of a vertex allows us to prove the following intuitive notion. Suppose the first $k$ vertices of an ordering $d$ already have the DPC property. Then we need only run the DPC algorithm along the remaining vertices of $d$ to enforce DPC on the entire graph. We formalise this in Lemma 5, where we use $S_{d_k}$ to denote the subnetwork induced by the vertices in $d_k$ only.

**Lemma 5.** Given an STN $S$ and an elimination ordering $d = (v_n, v_{n-1}, \ldots, v_1)$ of which the vertices $d_1 = (v_n, v_{n-1}, \ldots, v_{k+1})$ have the DPC property along $d$. Then running the DPC algorithm along $d_2 = (v_k, v_{k-1}, \ldots, v_1)$ in $S_{d_2}$ makes $S$ d-DPC.

**Proof.** Because $d_2$ is a subsequence of $d$, it follows from Corollary 1 that $d_2$ is a simplicial elimination ordering of
such that PPC on an STN extended with a new event enforces PPC on an STN extended with a new event $d$.

Theorem 1. Let $S'$ be an STN as just defined. Given a simplicial elimination ordering $d = (v_0, v_1, v_2, \ldots, v_n)$ of $S'$ such that $\{v_0, v_1, \ldots, v_k\} = N(a)$ and $v_1 = a$. Then, running the DPC algorithm along $d'$ makes $S'$ d-DPC.

Proof. It suffices to show that each vertex $v_k \in (v_0, v_1, \ldots, v_{\delta(a)+2})$ has the DPC property along $d$ in $S'$. The remainder of the proof then follows from Lemma 5. Assume for a contradiction that one such vertex is not d-DPC; then there must be some $v_i, v_j$ such that $i, j < k$ and $v_i, v_j \in N(v_k)$, but $w_{i\rightarrow j} > w_{j\rightarrow i}$, where $w_{i\rightarrow j}$ is the weight of the edge from $v_i$ to $v_j$. Since $S$ was already DPC and none of the weights in $S$ have changed, this inequality cannot hold if $v_i, v_j$ and $v_k$ are all in $S$. Hence one of them must be $a$, the only vertex not in $S$. Since $k > \delta(a) + 1$ and $a = v_1$, $v_k$ cannot be $a$. Furthermore, since $k > \delta(a) + 1$ and all neighbours of $v_k$ have $\ell \leq \delta(a) + 1$, $v_k$ cannot be a neighbour of $a$. But since $v_i$ and $v_j$ are neighbours of $v_k$, this means that $v_i$ and $v_j$ cannot be either, a contradiction. Thus, vertices $(v_0, v_1, \ldots, v_{\delta(a)+2})$ must indeed be d-DPC in $S'$.

4 Vertex-incremental PPC

In our new algorithm, we apply the observations on DPC from the previous section to present an algorithm that re-enforces PPC on an STN with a new event $a$ and associated new constraints $C_a$. It is possible that these new constraints $C_a$ break the chordality of the original STN $S$. We therefore require that chordality be re-enforced on the network in a pre-processing step before we run Vertex-IPPC. We end this section with the discussion of an algorithm by Berry et al. (2006) which accomplishes this.

After pre-processing, we use the Lex-BFS algorithm to find a simplicial elimination ordering ending in $a$. Since the network is chordal, this ordering is guaranteed to exist by Lemma 2. Now we know by Theorem 1 that DPC may only be violated in the subgraph induced by $a$ and its direct neighbours $N(a)$. Hence, it suffices to run the DPC algorithm in this subgraph to re-enforce DPC in the entire graph.

Algorithm 2: Vertex-IPPC

Input: A chordal STN $S'$ obtained by extending the PPC STN $S$ with (1) a new event $a$, (2) a set of constraints connecting $a$ to the events in $U \subseteq V$ with weights $C_a = \{w_{a\rightarrow u}, w_{u\rightarrow a} \mid u \in U\}$, and (3) a set of constraints with infinite weight needed to make $S'$ chordal.

Output: INCONSISTENT if the new constraints yield inconsistency, CONSISTENT if PPC has been enforced on the modified network $S'$.

1. $d \leftarrow$ Lex-BFS $(S', a)$
2. call DPC $(S' \setminus \{a\}, w_{a\rightarrow a}, d)$
3. return INCONSISTENT if DPC did
4. call SSSP-PPC $(S', d, a)$
5. return CONSISTENT

Algorithm 3: SSSP-PPC($S, d, a$)

Input: A PPC STN $S = (V, E)$ which is PPC on $S \setminus \{a\}$ and has associated weights $\{w_{i\rightarrow j}, w_{j\rightarrow i} \mid \{i, j\} \in E\}$, and a simplicial elimination ordering $d = (v_0, v_1, v_2, \ldots, v_n, v_{\delta(a)+2})$.

Output: The PPC network $S'$.

1. $D_{a\rightarrow a}[a] \leftarrow 0$; $D_{a\rightarrow a}[a] \leftarrow 0$
2. VISITED[a] $\leftarrow$ TRUE
3. for $k \leftarrow 2$ to $n$ do
4. $D_{a\rightarrow v_k}[v_k] \leftarrow \infty$; $D_{a\rightarrow [v_k]}[v_k] \leftarrow \infty$
5. call Visit($v_k$)

Since the only change is the introduction of a new event, we know that if a constraint $c_{ij}$ needs to be tightened, the new shortest path must contain the newly added vertex $a$. We therefore maintain single-source shortest paths from $a$ to every vertex and single-sink shortest paths from every vertex $a$ and tighten all constraints in one outward sweep along $d$. Since at this point the network is already PPC, we can use a slightly modified version of the SSSP procedure used by Planken et al. (2010) and referred to here as PPC-SSSP.

The original PPC-SSSP algorithm calculates the single-source/sink shortest paths (SSSP) distances from a single vertex $a$ to every other vertex in the graph and vice versa. By exploiting the knowledge that the graph is already PPC, PPC-SSSP visits every edge in the graph only twice, achieving a run-time bound of $O(mc)$.

Since any new shortest path from $i$ to $j$ must go through $a$, it must be of the form $i \rightarrow a \rightarrow j$. Moreover, PPC-SSSP calculates exactly the weights of these paths $i \rightarrow a$ and $a \rightarrow j$ for all $i$ and $j$. Thus we can re-enforce partial path consistency in the same sweep. Our modified algorithm, dubbed PPC-SSSP, is shown in Algorithm 3.

We prove the correctness of PPC-SSSP in two steps. First, in Lemma 6 we adapt the correctness proof of PPC-SSSP to show that PPC-PPC does in fact calculate the correct weights for the shortest paths between $a$ and any other ver-
\begin{align*}
\text{Procedure Visit}(v \in V) & \quad & \\
1 & \text{foreach } u \in N(v) \text{ such that } \text{VISITED}[u] = \text{TRUE} \text{ do} & \\
2 & \quad D_{a \rightarrow v} & \leftarrow \min\{D_{a \rightarrow v}, D_{\bar{a} \rightarrow v} + w_{u \rightarrow v}\} & \\
3 & \quad D_{\bar{a} \rightarrow v} & \leftarrow \min\{D_{\bar{a} \rightarrow v}, w_{v \rightarrow u} + D_{a \rightarrow u}\} & \\
4 & \quad \text{VISITED}[v] & \leftarrow \text{TRUE} & \\
5 & \text{foreach } u \in N(v) \text{ such that } \text{VISITED}[u] = \text{TRUE} \text{ do} & \\
6 & \quad w_{v \rightarrow u} & \leftarrow \min\{w_{v \rightarrow u}, D_{a \rightarrow v} + D_{\bar{a} \rightarrow [v]}\} & \\
7 & \quad w_{v \rightarrow u} & \leftarrow \min\{w_{v \rightarrow u}, D_{\bar{a} \rightarrow v} + D_{a \rightarrow [v]}\} & \\
\end{align*}

This information is then used in Lemma 7 to show that a run of SSSP-PPC enforces partial path consistency on the network we encounter in line 4 of Vertex-IPPC.

\textbf{Lemma 6.} When \text{VISITED}[v] is \text{TRUE}, \(D_{a \rightarrow [v]}\) and \(D_{\bar{a} \rightarrow [v]}\) are set to the total weight of the shortest paths from \(v\) to \(a\) and from \(a\) to \(v\) respectively.

\textbf{Proof.} The proof is by induction along the simplicial construction ordering \(d^{-1} = (v_1, v_2, \ldots, v_n)\) with \(v_1 = a\). For the base case, \(D_{a \rightarrow [a]}\) and \(D_{\bar{a} \rightarrow [a]}\) are set correctly in line 1 of SSSP-PPC. Now, for the induction step, assume that the weight \(D_{a \rightarrow [v]}\) is set correctly for all \(v \in \{v_1, \ldots, v_k\}\); we show that SSSP-PPC sets the correct weight for \(D_{a \rightarrow [v_{k+1}]}\). The proof for \(D_{\bar{a} \rightarrow [v_{k+1}]}\) is analogous.

Assume for a contradiction that \(\text{VISITED}[v_{k+1}]\) is \text{TRUE} but that \(D_{a \rightarrow [v_{k+1}]}\) does not hold the weight of shortest path. Then there must be some path \(\pi = a \rightarrow v_j \rightarrow \cdots \rightarrow v_{j_l} \rightarrow v_{k+1}\) with weight \(w_{\pi} < D_{a \rightarrow [v_{k+1}]}\). If there is a vertex \(j_{max} = \max_{j} j_{j_{i}}\) on \(\pi\) such that \(j_{max} > k\), we know by DPC that we can replace this vertex by a shortcut from its predecessor to its successor, without increasing the weight. By repeating this procedure we can remove all such \(v_j\) from the path \(\pi\) without increasing its total weight.

In particular, this means that \(j_{\ell} \leq k\), which implies that \(\text{VISITED}[v_{k+1}]\) is \text{TRUE} and therefore, by the induction hypothesis, that \(D_{a \rightarrow [v_{k+1}]}\) has been correctly set. But then the value of \(D_{a \rightarrow [v_{k+1}]}\) is correctly updated by the assignment on line 2 of \text{Visit}(v_{k+1}), contradicting our assumption that \(w_{\pi} < D_{a \rightarrow [v_{k+1}]}\).

\textbf{Lemma 7.} Given a \(d\)-DPC STN \(S\), where \(a\) is the last vertex of \(d\) and \(S \setminus \{a\}\) is PPC. Then the network \(S'\) produced by running SSSP-PPC on \(S\) along the reverse of \(d\) is PPC.

\textbf{Proof.} Consider the edges adjacent to \(a\), i.e. \(\{a, v\}\) with \(a \in N(v)\). Before the second loop of \text{Visit}(v) is executed, \(\text{VISITED}[a]\) and \(\text{VISITED}[v]\) are both \text{TRUE}, by lines 2 of SSSP-PPC and 4 of \text{Visit}, respectively. Hence, by Lemma 6, \(D_{a \rightarrow [v]}\) and \(D_{\bar{a} \rightarrow [v]}\) hold the weights of the corresponding minimum paths between \(v\) and \(a\). So the weights of \(w_{a \rightarrow v}\) and \(w_{v \rightarrow a}\) are correctly updated in lines 6 and 7 of \text{Visit}.

Now consider any other edge \(\{u, v\}\) neither endpoint of which is \(a\). Since \(S \setminus \{a\}\) was PPC, the only way in which the weight \(w_{u \rightarrow v}\) can be further reduced is if there is some shorter path \(u \rightarrow a \rightarrow v\) through \(a\). Without loss of generality, assume that \text{Visit}(u) is called before \text{Visit}(v). Then \(\text{VISITED}[u]\) and \(\text{VISITED}[v]\) are both \text{TRUE} before the execution of the second loop in \text{Visit}(v). By the same reasoning as used above for \(a\) and its neighbours, line 6 of \text{Visit} correctly updates \(w_{u \rightarrow v}\). The proof for \(w_{v \rightarrow u}\) is analogous, reversing all arcs and using line 7 of \text{Visit}(v) instead.

\textbf{Correctness and efficiency}

Having shown the correctness of the individual steps of the Vertex-IPPC algorithm, we now complete our analysis by demonstrating that these steps do indeed re-enforce partial path consistency when executed in sequence.

\textbf{Theorem 2.} Vertex-IPPC correctly re-enforces PPC or decides inconsistency in \(O(m_c + \delta_c(a) w_d^2)\) time.

\textbf{Proof.} The simplicial elimination ordering \(d\) of \(S'\) is obtained using Lex-BFS, and therefore, by Lemma 2, the last vertices in \(d\) are the neighbours of \(a\), followed by \(a\) itself as final vertex. Since \(S\) was PPC, by Theorem 1 the call to DPC on the sub-graph consisting of \(a\) and its neighbours re-enforces DPC along \(d\) for the new \(S'\), or concludes inconsistency. Finally, by Lemma 6, running SSSP-PPC along \(S'\) re-enforces PPC on \(S'\).

We now turn to the time complexity. Lex-BFS takes \(O(m_c)\) time to construct the ordering \(d\), and running DPC on an induced graph with \(\delta_c(a) + 1\) nodes takes \(O(\delta_c(a) w_d^2)\) time. Finally, SSSP-PPC takes \(O(m_c)\) time: Visit is called once per vertex and all operations in Visit take amortised constant time per edge.

\textbf{Incremental triangulation}

So far we have focused exclusively on the correctness of Vertex-IPPC. We now discuss an important advantage of our algorithm over Edge-IPPC: it integrates well with an incremental triangulation algorithm. Recall that Edge-IPPC requires that the edge \((a, b)\), representing the constraint \(c_{a \rightarrow b}\) to be tightened, is already part of the underlying structure. If the edge is not present, a new triangulation must be found in which it does exist. We cannot simply include \((a, b)\), since it may introduce a new chordless cycle. This would break the chordality required for PPC algorithms.

For Vertex-IPPC, we can address this issue by using a recently discovered algorithm by Berry et al. (2006). This algorithm performs vertex-incremental minimal triangulation: given a chordal graph \(G = (V, E)\), a new vertex \(a\), and a set of edges \(E_a\) connecting \(a\) to existing vertices of \(G\), it can compute the minimal fill \(F\) such that the graph \(G' = (V \cup \{a\}, E \cup E_a \cup F)\) is chordal.

Their algorithm needs \(O(n)\) time to insert one edge, and since there are \(m\) edges it takes \(O(mn)\) time to triangulate a graph from scratch. An interesting observation here is that if we average this time over the number of vertices, we obtain an amortised bound of \(O(m)\) time to insert one vertex. Since \(m = O(m_c)\), this fits comfortably in the \(O(m_c + \delta_c(a) w_d^2)\) bound for one run of Vertex-IPPC.

We learned from the authors that they were unaware of any existing implementation of this algorithm. To evaluate our algorithm, we therefore had to implement it ourselves. This turned out to be not exactly trivial, but after having found an efficient way to maintain references to the cliques in the maintained clique-tree, we were able to stay within the established bounds (ten Thije, 2011).
5 Empirical evaluation

Having established the theoretical soundness and efficiency of our algorithm, we now turn to assess its performance in practice. In particular, we want to answer two questions. First, does our implementation meet the run time bound of $O(m_c + \delta_c(a)w_a^2)$ derived in Theorem 2, and second, how does its performance compare to Edge-IPPC?

Regarding the second question, comparing a vertex-incremental algorithm to an edge-incremental one may seem strange. However, we observe that Edge-IPPC can be used to enforce PPC after a vertex has been inserted, using the following simple approach. After the graph has been pre-processed such that it contains $a$ and will still be chordal after all new constraints are added, we simply run Edge-IPPC once for all constraints ($\delta(a)$ in number) attached to $a$. Since each run of Edge-IPPC takes $O(m_c)$ time, this sequential algorithm re-enforces PPC in $O(\delta(a) \cdot m_c)$ time.

We expect that Vertex-IPPC outperforms this sequential method. The main reason is that $w_a^2$ is the number of edges in the largest clique, which is usually far smaller than $m_c$, the number of edges in the entire chordal graph. While $\delta(a)$ is smaller than $\delta_c(a)$, we expect this will have only little effect. After all, increasing $\delta_c(a)$ only results in more work in the new clique containing $a$, while increasing $\delta(a)$ yields more runs spanning the entire graph.

Hypothesis 1 and Hypothesis 2 present the expected answers to respectively the first and second question posed above.

Hypothesis 1. The run time of our implementation of Vertex-IPPC meets the theoretical bound of $O(m_c + \delta_c(a)w_a^2)$.

Hypothesis 2. When inserting a new vertex $a$ in a graph, running Vertex-IPPC is faster than running Edge-IPPC $\delta(a)$ times.

Data sets

Since we aim to assess asymptotic behaviour, we require graphs with large numbers of vertices. Moreover, since we concern ourselves with problems related to scheduling, we would like input graphs from this domain as well.

We therefore used a generator for STNs based on Hierarchical Task Networks (HTNs), a planning paradigm formalised by Erol et al. (1994). This generator was used by Planken et al. (2010) to empirically evaluate the Edge-IPPC algorithm. We briefly summarize their discussion of HTNs; for more details we refer to the original papers.

In short, an HTN represents a hierarchy of tasks, in which high-level tasks are progressively decomposed into collections of smaller tasks as we go further down the hierarchy. A general HTN can contain multiple different such decompositions of the same task. However, such choices are beyond the expressive power of STNs. We therefore only consider HTNs whose tasks can be decomposed in at most one way.

Constraints in an HTN occur only between a parent task and its children, or between sibling tasks. The latter constraints are called sibling-restricted (SR) edges. Given this restriction, it is not possible to coordinate the execution of tasks in the different branches of the hierarchy when we strictly adhere to the HTN format. In order to circumvent this, the definition of an HTN may be extended to include so-called landmark variables (Castillo et al., 2006), which do allow synchronisation between branches. Finally, each task is modelled as a pair of events in the corresponding STN: its start and its end.

In the interest of reproducibility, the settings we used to generate our HTNs are listed in Table 1. Using these settings, each generated graph contained 220 vertices. Our benchmark set consisted of 80 such graphs.

To assign the weights, we used a utility provided with the HTN generator by Planken et al. (2010). We configured this utility to first ensure a solution exists, by assigning a random time between $-50$ and $100$ to each event. Then, the weight for each constraint in the network was calculated as follows. First, the utility determined the minimum possible weight such that the random solution was still possible. It then increased this weight by a value selected randomly from the range $[0, 150]$. Hence the original solution still existed, but had been “obscured” by the randomly added slack.

In order to test the influence of the graph structure on performance, we also generated a set of chordal graphs. This set was also created with the HTN generator and the weight setting utility. However, for this set we first added the fill edges as inserted by the minimum degree heuristic, before assigning weights.

Finally, we used sets of job-shop and scale-free graphs which the original Edge-IPPC algorithm was benchmarked on. We refer to Planken et al. (2010) for a discussion of the properties of these graphs.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Branch depth [min,max]</th>
<th>Branches [min,max]</th>
<th>Landmark ratio</th>
<th>Probability of SR-edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>[3, 6]</td>
<td>[3, 10]</td>
<td>0.20</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Table 1: Parameters for generating HTN graphs.

Method

In order to assess performance, we used the following procedure. For each experiment we incrementally constructed three graphs: two graphs used to benchmark Vertex-IPPC and Edge-IPPC, and one control graph on which we ran the P^C algorithm to verify that our implementation was correct.

All three graphs were initially empty. During the experiment we inserted the vertices from the input graph into the three experiment graphs one by one, as well as all constraints connecting the new vertex to the other vertices already in the graph. The order in which vertices were inserted was chosen randomly. When a vertex and its constraints were inserted, we first used the vertex-incremental triangulation scheme by Berry et al. (2006) to ensure the resulting graph was chordal. We then re-enforced partial path consistency by three different methods. On the first graph we ran Vertex-IPPC, starting from the newly added vertex. On the second graph, we ran Edge-IPPC for each constraint attached to the new vertex $a$,...
for a total of $\delta(a)$ runs. Finally, we ran P3C on the third graph.

At the end of each step, we made sure the weights in the graphs maintained by Vertex- and Edge-IPPC were equivalent to the weights enforced by P3C, thus verifying that the implementation was correct.

**Results and discussion**

**Verification of theoretical bounds** The aim of our first experiment was to verify our implementation against the theoretical upper bound of $O(m_c + \delta_c(a)w^2_d)$ derived in Theorem 2. We therefore plot the run time required to insert a vertex as function of the complexity of the graph at the time of insertion. By taking $x = m_c + \delta_c(a)w^2_d$ as measure of complexity, we should see a straight line if our implementation meets this bound on the given input graphs. Fig. 1 shows this plot for the normal HTN graphs, and we do indeed observe a line with a clear linear trend. In other words, our experiments support Hypothesis 1 and we conclude that we cannot reject it.

**Comparison to Edge-IPPC** In our second experiment we compare the performance of using Vertex-IPPC to insert a vertex at once, to the performance of running Edge-IPPC once for every constraint adjacent to the new vertex. We initially ran this experiment on the set of normal HTN graphs and obtained the results summarised in Fig. 2.

Rather surprisingly, we find that Edge-IPPC is significantly faster than Vertex-IPPC in this benchmark. The results for the job-shop and scale-free graphs were similar, so in the interest of saving space we omit these graphs. Hence, Hypothesis 2 does not hold in general, and we reject it.

A possible explanation for this somewhat disappointing result is that the degrees of the new vertices in the chordal graph may have been much higher than their degrees in the original graph. Recall that Edge-IPPC needs to be executed only once for each constraint in the original graph, and completes in time linear in the number of edges. On the other hand, the run time of the DPC algorithm used by Vertex-IPPC depends on the size of the neighbourhood in the triangulated graph. Running DPC is relatively expensive, so if there are only a few real constraints but many fill edges, this investment may not pay off.

In particular, we may have the following pathological case. Suppose the new vertex $a$ has only two constraints involving older vertices in the original graph. However, these constraints create a cycle involving all $n$ vertices, and thus after triangulation, $a$ is connected to all vertices in the chordal graph. In this case we need only run Edge-IPPC twice, which can be done in $O(m_c)$ time, while the DPC step alone takes $O(nw^2_d)$ time. Conversely however, if nearly all edges in the chordal graph correspond to actual constraints, the Edge-IPPC method will take $O(nm_c)$ time, which is worse than the $O(nw^2_d)$ that dominates the run time of Vertex-IPPC in this case.

We summarise the above in the following new hypothesis, a more specific form of Hypothesis 2:

**Hypothesis 3.** Given that $\delta(a) = \delta_c(a)$ when a new vertex $a$ is inserted in a graph, running Vertex-IPPC is faster than running Edge-IPPC $\delta(a)$ times.

To test this hypothesis, we ran our experiment on the input set consisting of chordal HTNs. Since these graphs are already triangulated, we do not need to add any fill edges, which should give Vertex-IPPC an advantage over Edge-
IPPC. The results for this experiment are shown in Fig. 3. This figure does indeed show a better relative performance for Vertex-IPPC, but the difference is small. Since Vertex-IPPC does not clearly perform better than Edge-IPPC, we have to reject Hypothesis 3.

A final possibility to explain these results is the following. It may be that the consecutive runs of Edge-IPPC in the neighbourhood of the new vertex “help” each other. More specifically, it may be that two constraints $c_1$ and $c_2$ imply a lower weight for some constraint $c_3$ than $c_3$ is itself labelled with in the source graph. If Edge-IPPC enforces $c_1$ and $c_2$ first, it can then immediately detect that the weight of $c_3$ need not change, thus reducing the execution cost. If this happens frequently, then Edge-IPPC essentially gets a few runs “for free”. Vertex-IPPC on the other hand always needs to pay the cost of running DPC, whether constraints subsume each other or not.

While this hypothesis does not offer a way to improve the performance of Vertex-IPPC, it can be used to improve that of Edge-IPPC. In particular, it would be interesting to see whether there is some way to put the constraints inserted by Edge-IPPC in such an order that the “help” is maximised. We will not pursue this idea here, but note it as an interesting direction for future work.

6 Conclusions and future work

We showed that DPC can be defined on subnetworks of an STN as well, and that running the DPC algorithm along the neighbours of an inserted vertex is sufficient to guarantee DPC on the complete graph. This idea is used in Vertex-IPPC, a new algorithm to re-enforce partial path consistency in an STN when it is extended with a new event and associated constraints. The run time of this algorithm is bounded by $m_{av} n$, the number of edges in the chordal graph and $n w_{av}^2$, where $n$ is the number of vertices in the network and $w_{av}$ is the network’s induced width.

Our algorithm integrates nicely with the vertex-incremental triangulation method recently presented by Berry et al. (2006). We implemented this triangulation method for our experiments, which, to the best of our knowledge, had not been done before.

Surprisingly, these experiments on HTN, job-shop and scale-free graphs show that Vertex-IPPC is outperformed by sequentially inserting the new constraints using the existing Edge-IPPC algorithm. On chordal networks, which offer the greatest advantage for Vertex-IPPC relative to Edge-IPPC, both algorithms show similar performance in practice. We conjecture that the sequential method may benefit from the order in which constraints are inserted. As part of our future work we aim to investigate this idea and determine if an optimal ordering can be found to further optimise the performance of Edge-IPPC.

Secondly, methods are under development to also deal with constraint loosening and edge deletions (ten Thije, 2011). 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.

Bibliography


