Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling

Distributed Algorithms for Incrementally Maintaining Multiagent Simple Temporal Networks

James C. Boerkoel Jr.[1]* Léon R. Planken[2]* Ronald J. Wilcox[1] Julie A. Shah[1]
[1] Computer Science and AI Laboratory, Massachusetts Institute of Technology, USA
[2] Faculty of EEMCS, Delft University of Technology, The Netherlands
{boerkoel, julie_a_shah}@csail.mit.edu l.r.planken@tudelft.nl rjwilcox@mit.edu

Abstract

When multiple agents want to maintain temporal information, they can employ a Multiagent Simple Temporal Network (MaSTN). Recent work has shown that the constraints in a MaSTN can be efficiently propagated by enforcing partial path consistency (PPC) with a distributed algorithm. However, new temporal constraints may arise continually due to ongoing plan construction or execution, the decisions of other agents, and other exogenous events. For these new constraints, propagation is again required to re-establish PPC. Because the affected part of the network may be small, one typically wants to exploit the similarities between the new and previous version of the MaSTN. To this end, we propose two new distributed algorithms for incrementally maintaining PPC. The first is inspired by △STP, the seminal PPC algorithm for STNs; the second is a distributed version of IPPC, which represents the current state of the art for incrementally enforcing PPC in a centralized setting. The worst-case time performance of these algorithms is similar to their centralized counterparts. We empirically compare our distributed algorithms, analyzing their performance under various assumptions, and demonstrate significant speedup over their centralized counterparts.

Introduction

Simple Temporal Networks (STNs) offer a way to efficiently maintain sets of temporal constraints. In many planning and scheduling domains, agents must coordinate with others while efficiently managing their own temporal constraints. Indeed, STNs have played a central role in many deployed planning systems with applications in the coordination of military and disaster relief efforts, Mars rover missions, health care operations, and manufacturing tasks (Laborie and Ghallab 1995; Bresina et al. 2005; Castillo et al. 2006; Barbulescu et al. 2010; Wilcox and Shah 2012).

The Multiagent STN (MaSTN; Boerkoel and Durfee 2013) enables agents, which were previously forced to use a single centralized STN, to capture their interacting temporal constraints in a *decentralized* manner. This representation allows each agent to maintain its local portion largely independently of other agents, leading to increased concurrency and privacy. Partial path consistency (PPC; Xu and Choueiry 2003; Planken, De Weerdt, and Van der Krogt 2008) provides a way for efficiently propagating temporal constraints while exploiting network sparsity, e.g., the loosely-coupled nature of an MaSTN. However, due to ongoing plan construction or execution, the decisions of other agents, or other exogenously determined events, new constraints can arise that invalidate partial path consistency. Recent work provides a (centralized) algorithm called IPPC for enforcing PPC incrementally, by exploiting the similarities between the new and previous versions of the temporal network (Planken, De Weerdt, and Yorke-Smith 2010).

In this paper, we apply insights from the IPPC algorithm to the distributed MaSTN representation to develop two new *distributed* algorithms for *incrementally* enforcing PPC. The first algorithm is inspired by △STP (Xu and Choueiry 2003), the seminal algorithm for enforcing PPC on STNs, and the second is a distributed version of the state-of-the-art centralized algorithm IPPC. They attempt to optimize the concurrent runtime of algorithms using two different strategies—the first attempts to maximize agent utilization, while the second attempts to minimize total effort. The worst-case time performance of these algorithms is similar to their centralized counterparts. However, based on key insights about the MaSTN, we demonstrate that distributed, concurrent computation is possible. Finally, we empirically compare our distributed algorithms, analyzing which algorithm performs best under various assumptions, and demonstrate significant speedup over their centralized counterparts.

Background

A *Simple Temporal Problem (STP)* (Dechter, Meiri, and Pearl 1991) instance consists of a set $X = \{x_1, \ldots, x_n\}$ of n 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 $c_{i\to j} \in C$ defines a value $b_{i\to j} \in \mathbb{R} \cup \{\infty\}$ corresponding to an upper bound on this difference, and represents an inequality $x_j - x_i \leq b_{i\to j}$. Two constraints $c_{i\to j}$ and $c_{j\to i}$ can be combined into a single constraint interval $x_j - x_i \in [-b_{j\to i}, b_{i\to j}]$, giving both upper and lower bounds. An unspecified constraint is equivalent to a constraint with an infinite weight; therefore, if $c_{i\to j}$

*These authors are listed in alphabetical order and contributed equally to this work during the European Extended Lab Visit Program funded by the NSF. A version of this work was presented at the 2012 Autonomous Robots and Multirobot Systems Workshop (ARMS 2012).

Copyright © 2013, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

exists and $c_{j\to i}$ does not, we have $x_j - x_i \in (-\infty, b_{i\to j}]$.

Each instance of the STP has a natural graph representation called a *Simple Temporal Network (STN)*. Because our algorithms can be stated more naturally using this representation, we use it throughout the remainder of this paper. In an STN $\mathcal{S} = \langle V, E\rangle$, each temporal variable is represented by a vertex $v_i \in V$, and each constraint is represented by an edge $\{v_i, v_j\} \in E$ between vertices v_i and v_j with two associated weights, $w_{i\to j}$ and $w_{j\to i}$, which are initially equal to $b_{i\to j}$ and $b_{j\to i}$, respectively. The continuous domain of each variable $v_i \in V$ is defined as a bound $[e, l]$ over the difference $v_i - z$, where z is a special zero time point representing the start of time and e and l represent v_i's earliest and latest times, respectively. To reduce clutter when depicting an STN, we often omit z and instead represent these constraints as unary ones, i.e., self-loops labeled by clock times.

The *Multiagent Simple Temporal Network* (Boerkoel and Durfee 2013) or MaSTN is informally composed of N sub-STNs, one for each agent A in a set $\{1, \ldots, N\}$, and a set of edges E_X that establish relations between the sub-STNs of different agents. V_L^A is defined as agent A's set of *local vertices*, which corresponds to all time-point variables assignable by agent A. E_L^A is defined as agent A's set of *local edges*, where a local edge $\{v_i, v_j\} \in E_L^A$ connects two local vertices $v_i, v_j \in V_L^A$. The sets V_L^A partition the set of all non-zero time points. Together, V_L^A and E_L^A form agent A's *local sub-STN*, $\mathcal{S}_L^A = \langle V_L^A, E_L^A\rangle$. Each *external edge* in the set E_X connects sub-STNs of different agents and is incident to two vertices that are local to different agents, $v_i \in V_L^A$ and $v_j \in V_L^B$, for $A \neq B$. Each agent A is aware of E_X^A, the set of external edges that involve exactly one of A's local vertices; formally: $E_X^A = \{\{v_i, v_j\} \in E_X \mid v_i \in V_L^A \wedge v_j \notin V_L^A\}$. Apart from its local vertices, agent A also knows about V_X^A, the set of non-local vertices involved in E_X^A. In summary, agent A is aware of its *known time points*, $V^A = \{V_L^A \cup V_X^A\}$, and its *known constraints*, $E^A = \{E_L^A \cup E_X^A\}$. The joint MaSTN $\mathbf{S}$ is then formally defined as the set of sub-STNs $\mathcal{S}^i = \langle V^i, E^i\rangle$ for $i \in \{1, \ldots, N\}$.

Example. Consider scheduling the activities of three agents—two mobile manufacturing robots (A,B) and a human quality control inspector (H)—in a manufacturing environment, displayed as STNs in the top, middle, and bottom rows of Figure 1, respectively. The robots must perform three manufacturing tasks D, F, and G, e.g., welding or torquing parts into place. The human inspector is responsible both for an inspection task I, and for conducting routine maintenance on robot B: task M. In this problem, each agent has various local constraints over when activities can occur, including task duration and transition times between tasks. For instance, the start and end events of task D are represented in Figure 1 as the vertices D_S^A and D_E^A, respectively; the constraint that D requires between 40 and 70 minutes is represented as an edge from D_S^A to D_E^A with label $[40, 70]$.

In addition, there are external constraints, represented as dashed lines, that establish relationships between the agents. In our example, while performing task D, robot A obstructs the route to the location where the maintenance M of robot B

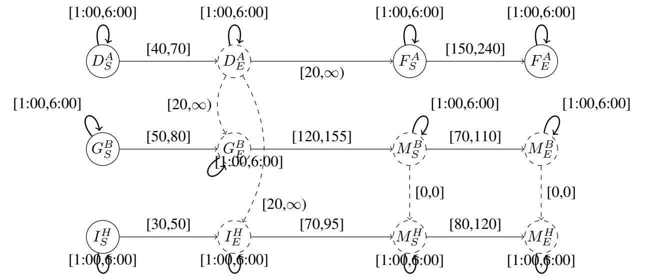


Figure 1: The interacting schedules of two manufacturing robots and a human inspector depicted in an STN.

must take place; thus, robot A must be allowed 20 seconds after the completion of task D to clear the way for robot B and the inspector. Further, in the inspector's and robot B's local representations of M, the start and end times must coincide exactly. Robot A's set of known time points includes the vertices in the top row of Figure 1 as well as G_E^B, and I_E^H; and its set of known edges includes all edges between these time points.

Solving the STP. Solving an STP is often equated with determining its set of solutions: those assignments of values to the variables that are consistent with all constraints. Since the size of a naïve representation of this solution set is prohibitive, we often instead compute the equivalent *minimal network* $\mathcal{M}$, where each constraint captures the exact set of temporal differences that will lead to solutions. $\mathcal{M}$ allows constant-time answering of queries such as (i) whether the information represented by the STP is consistent; (ii) finding all possible times at which some event x_i could occur; and (iii) finding the minimum and maximum temporal difference between two events x_i and x_j implied by the set of constraints. For the STP, the minimal network can be found by enforcing *path consistency*, or equivalently by computing all-pairs shortest paths on the associated STN, which yields a complete graph and requires $\mathcal{O}\left(n^3\right)$, $\mathcal{O}\left(n^2 \log n + nm\right)$ or $\mathcal{O}\left(n^2 w_d\right)$ time depending on the algorithm (Planken, De Weerdt, and Van der Krogt 2011), where $n = |V|$, $m = |E|$, and w_d is described below.

Instead of enforcing PC on an STN $\mathcal{S}$, one can opt to enforce *partial path consistency* (PPC; Bliek and Sam-Haroud 1999) to yield a potentially much sparser *chordal* or *triangulated* network $\mathcal{M}^*$, where every cycle of length four or more has an edge joining two non-adjacent vertices in the cycle. As in $\mathcal{M}$, all edges $\{v_i, v_j\}$ in $\mathcal{M}^*$ are labeled 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. Thus, $\mathcal{M}^*$ shares $\mathcal{M}$'s properties of equivalence to $\mathcal{S}$, constant-time resolution of the queries mentioned, and efficient, backtrack-free extraction of any solution. The main drawbacks of a PPC network as compared to its PC counterpart are that (i) it cannot directly resolve queries involving arbitrary pairs of variables (i.e., those not connected in the chordal graph), and (ii) updates to the network cannot be directly propagated through the fully-connected network, but rather require traversing the chordal network in a particular way, as we describe later. The number of edges in $\mathcal{M}^*$,

denoted by m_c, is bounded by $\mathcal{O}(nw_d)$. Here, w_d is the *graph width* induced by an ordering d of the vertices. P^{3}C, regarded as the current state of the art for solving an STN non-incrementally, tightens triangles in a principled manner to establish PPC in $\mathcal{O}(nw_d^2)$ time (Planken, De Weerdt, and Van der Krogt 2008). Other recent algorithms similarly exploit network structure to propagate constraints in sparse temporal networks (e.g., Xu and Choueiry 2003; Bui, Tyson, and Yorke-Smith 2008; Shah and Williams 2007).

Distributed and Incremental Approaches. Boerkoel and Durfee (2010) present DP^{3}C, a distributed version of the P^{3}C algorithm, in which agents independently process as much of their local problem as possible and coordinate to establish PPC on the external portions of the MaSTN. While DP^{3}C has the same worst-case time complexity as P^{3}C, Boerkoel and Durfee show that due to concurrent execution, DP^{3}C achieves significant speedup over P^{3}C, especially when the relative number of external constraints is small.

The incremental partial path consistency algorithm (IPPC; Planken, De Weerdt, and Yorke-Smith 2010) takes an already PPC network and a constraint $c_{a\rightarrow b}$ to be tightened (i.e., an edge $\{v_a, v_b\}$ whose weight is to be lowered). It is based on the idea that, in order to maintain PPC, the weights of edges in a chordal graph only need to be updated if at least one of the neighbors has an incident edge that is updated. It runs in $\mathcal{O}(m_c)$ and $\mathcal{O}(n_c^*\delta_c)$ time and $\mathcal{O}(m_c)$ space. Here, n_c^* and δ_c are the number of endpoints of updated edges in the chordal graph and the chordal graph's degree, respectively.

Example Revisited. Figure 2 represents a PPC version of the STN from Figure 1, where bounds (in black) have been tightened to specify the exact set of feasible values, and three inter-agent edges have been added between B and H to triangulate the graph. Suppose, however, that the inspector finds out that an unexpected event causes the minimum transition time between her inspection and maintenance tasks to increase from 70 to 80 minutes. In Figure 2, the results from this update are depicted by striking out old bounds and replacing them by newly updated ones in red. As demonstrated, only a small portion of the network (the red, double edges), needs to be revisited to re-establish PPC.

Figure 2: Only a small portion (red) of the PPC network of the example problem must be revised when the minimum transition time is updated to 80 seconds.

Distributed Incremental △STP

Xu and Choueiry (2003) were the first to recognize that PPC could be established on STNs. Their algorithm △STP forms a queue of all triangles (any triplet of pairwise connected time-point variables), establishes PC on each triangle in the queue, and re-enqueues all triangles containing an updated edge.

The idea of creating an incremental, distributed version of the △STP algorithm is relatively straightforward: partition the set of all triangles in the MaSTN among agents and have each agent independently maintain its own separate, local triangle queue. To achieve distribution, any time an agent updates an edge that is shared with some other agent(s), it must communicate the update to them. The main tweak for incrementalizing the algorithm is that each agent initializes an empty local queue and enqueues triangles incident to each updated edge (received from either "nature" or another agent, or after making a scheduling decision itself). Agents continue processing their queues of triangles and exchanging messages until quiescence is reached. In this work, we investigate processing only one update at a time. However, this approach could potentially process many asynchronous updates concurrently, though there is no guarantee that the set of asynchronously introduced updates will be collectively self-consistent.

An important issue is how to partition responsibility for maintaining and communicating triangle edge information among agents. Fortunately, we can exploit the extant DP^{3}C algorithm to triangulate and establish PPC on the MaSTN as a preprocessing step. This leads to a natural policy for assigning the responsibility for each triangle to exactly one agent—the agent that created or realized the triangle by being the first to eliminate one of its time points. For instance, in Figure 2, triangle $\{G_E^B, I_E^H, z\}$[1] is created when agent B eliminates G_E^B, so under our policy agent B would be responsible for making sure this triangle is self-consistent. Additionally, because of the way agents construct a global elimination order, triangles tend to be naturally load-balanced among all agents, though privacy concerns dictate that an agent can only assume responsibility for triangles that contain one of its vertices. In our example, agents A, B, and H are responsible for 6, 8, and 7 triangles respectively, where $\{G_E^B, I_E^H, z\}$ becomes B's responsibility even though A, which has no vertices involved, has fewer triangles.

We present pseudocode for the Distributed Incremental △STP algorithm (DI△STP) as Algorithm 1. Initially, each agent is assumed to know which triangles it is responsible for and constructs its own empty, local queue. Then, agent i waits until it receives an update message, checks if this results in a local update that is tighter than its current network[2], and if so, adds any local, incident triangles to its queue. Otherwise, the agent processes a triangle on the queue by testing whether each pair of edges implies a tighter weight for the third edge or not. If the agent does in fact update the edge weight, it adds all local incident triangles to its queue and sends the update to all neighboring agents. Termination occurs when the algorithm reaches quiescence. Asymptotically, DI△STP

[1] We list triangle vertices in elimination order to clarify which agent is responsible for each triangle: the owner of the first vertex.

[2] We write x.update(y) as shorthand for $x \leftarrow \min\{x, y\}$.

Algorithm 1: Distributed Incremental $\triangle$STP

Input: The triangles of agent i's local PPC MaSTN
Output: An updated PPC MaSTN
$Q_\triangle \leftarrow$ new, empty queue of triangles
while $Q_\triangle$.size() > 0 **or** PENDINGEDGEUPDATES() **do**
  while $\left(w'_{j\to i}, w'_{i\to j}\right) \leftarrow$RECEIVEUPDATE() **do**
    $w_{i\to j}$.update($w'_{i\to j}$); $w_{j\to i}$.update($w'_{j\to i}$)
    if an edge weight changed **then**
      $Q_\triangle$.ADDINCIDENTTRIANGLES($\{v_i, v_j\}$)
  $\{v_a, v_b, v_c\} \leftarrow Q_\triangle$.PEEK()
  foreach permutation (i, j, k) of $\{v_a, v_b, v_c\}$ **do**
    $w_{i\to j}$.update($w_{i\to k} + w_{k\to j}$)
  foreach updated edge $\{v_i, v_j\}$ **do**
    $Q_\triangle$.ADDINCIDENTTRIANGLES($\{v_i, v_j\}$)
    foreach agent A s.t. $\{v_i, v_j\} \in E^A$ **do**
      SENDUPDATE($A, (w_{j\to i}, w_{i\to j})$)
  $Q_\triangle$.REMOVE($\{v_a, v_b, v_c\}$)
return S^i

requires no more time to run than the original $\triangle$STP algorithm: in the worst case, all triangles may be affected by an update and belong to a single agent. However, since edge weights only decrease, the DI$\triangle$STP algorithm is guaranteed to converge to a fixed point without oscillation and in a finite number of steps. In practice, we expect that the asynchronous, concurrent nature of DI$\triangle$STP will lead to significantly better performance.

Next, we discuss how DI$\triangle$STP propagates the update in Figure 2. The updated edge $I^H_E - M^H_S \in [80, 95]$, leads to agent H placing two triangles, $\{I^H_E, M^H_S, z\}$ and $\{M^H_S, G^B_E, I^H_E\}$, on its queue. Agent H's processing of the first of these triangles leads to the edge update $I^H_E - z \in$ [2:15,3:20], which is communicated to agents A and B, who share knowledge of the edge. This leads to the addition of $\{D^A_E, I^H_E, z\}$ to A's queue, $\{G^B_E, I^H_E, z\}$ to B's queue, and $\{I^H_S, I^H_E, z\}$ to H's queue. Each agent proceeds to update the next triangle on its queue, which in turn leads to edge updates $D^A_E - I^H_E \in [45, 100]$ (by A) and $G^B_E - I^H_E \in [25, 75]$ (independently by both B and H). After these edge updates are properly communicated and processed, agent A (whose triangle $\{D^A_E, G^B_E, I^H_E\}$ leads to no new updates) and agent H (which computes an edge update, $I^H_S - z \in$ [1:35,2:50], that is not incident to any other triangles), finish processing their queues, which terminates the algorithm.

DI$\triangle$STP has properties that lead to various computational trade-offs. Non-local effects of an update are propagated to other agents quickly, which allows each agent to start working immediately, with the possibility of also terminating earlier. If the effect of an update is only local in scope, an agent naturally completes the update independently of the others. If the update affects more than one agent, DI$\triangle$STP exploits the inherent load-balancing of triangles that occurs as a result of distributed triangulation. The algorithm is asynchronous, which allows agents to maximize independence and autonomy in updating their local STNs and retain the privacy properties achieved by DP^{3}C . We thus expect that this algorithm will do well at maximizing agent utilization (and minimizing agent idle time). However the downside of an agent that optimistically and immediately processes its triangle queue is that it may do so using stale edge information, requiring later reprocessing. Indeed, like the original $\triangle$STP algorithm, DI$\triangle$STP may reprocess the same triangle many times; in pathological cases, it may even require effort quadratic in the number of triangles (Planken, De Weerdt, and Van der Krogt 2008), though, as mentioned, it will always converge. Next, we describe our Distributed IPPC algorithm that attempts to address this downside by traversing the temporal network in an explicitly principled order.

Distributed IPPC

DIPPC, our algorithm for distributed incremental partial path consistency, builds on the centralized IPPC algorithm (Planken, De Weerdt, and Van der Krogt 2011), which tags every vertex v in an order found through Maximum Cardinality Search (MCS; Tarjan and Yannakakis 1984), yielding an ordering of vertices in the chordal graph with minimum induced width w_d: a *simplicial construction ordering*. IPPC's main addition to MCS is that as each vertex v is visited, arrays $D^{\downarrow}_a[v]$ and $D^{\uparrow}_b[v]$ are used to maintain, respectively, the length of the shortest path to a and from b, where $\{a, b\}$ is the new constraint edge. The tag procedure uses these arrays to update edge weights between v and each of previously tagged neighbor u, checking if there is a shorter path from u to v via both a and b.

For DIPPC, we further make use of a *clique tree*. For every chordal graph, an equivalent clique tree representation can be found efficiently (in linear time) using the same distributed triangulation preprocessing step as the DI$\triangle$STP algorithm. While both algorithms operate on the same underlying chordal graph, the DI$\triangle$STP algorithm treats *triangles* as first-class objects, whereas DIPPC treats *cliques* (the collection of triangles formed by a fully-connected subgraph) as first-class objects. Clique tree nodes have a one-to-one correspondence to the maximal cliques in the chordal graph. The clique tree representation, then, is an abstraction of the underlying chordal constraint network, that is guaranteed to be no larger than the original graph, whereas the triangle graph used by DI$\triangle$STP may require up to n^3 space for dense graphs.

The key innovation in our DIPPC algorithm is that, whereas IPPC followed an MCS ordering, we instead observe that the algorithm is correct when following *any* simplicial construction ordering starting from a tightened edge. Thus, we can set the node whose associated clique contains both endpoints of the tightened edge $\{a, b\}$ as the root of the clique tree. A traversal of the clique tree—where a parent node is visited before any of its children—then corresponds to a simplicial construction ordering of the chordal graph. The tree structure of the clique tree allows propagation to branch to other agents and so achieve concurrency.

Observation. *For re-enforcing PPC, vertices can be tagged in any order corresponding to a traversal of the clique tree,*

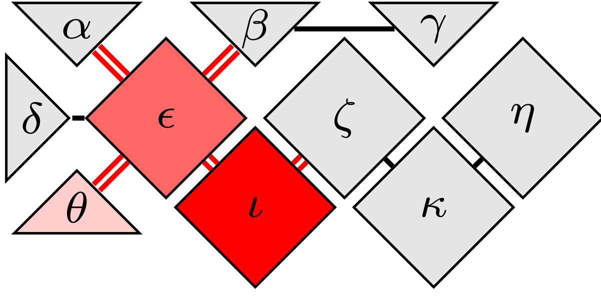


Figure 3: Clique tree representation of the example network.

starting at a clique containing the updated edge.

Consider again the example network from Figure 2 and its associated clique tree included in Figure 3. Every clique contains the temporal reference point z (used to reason about absolute time). Notice there are ten maximal cliques, and due to the implicit edge that all vertices share with z, all maximal cliques are of size 3 or 4 as denoted by triangles or diamonds, respectively. Each agent holds a copy of the part of the clique tree that contains its own vertices and the adjacent clique tree nodes. Furthermore, each clique is designated to be owned by the agent who first eliminates a vertex in that clique, like the triangles for DI△STP. In our example, the initial update occurs in clique ι, which consists of vertices I_E^H, G_E^B, M_S^H and z. Clique ι thus serves as the clique tree's root, and the inspector's agent, who is responsible for maintaining it, kicks off DIPPC, presented as Algorithm 2.

While the careful bookkeeping—done through the LIVE and PROP messages—makes the DIPPC algorithm appear complex, the actual conceptual flow of network updates—through the TAG messages and the MakeLive procedure—follows the original IPPC closely. As in IPPC, shortest distances $D_a^{\downarrow}[v]$ and $D_b^{\uparrow}[v]$ are maintained for every vertex v while propagating the change. Before every tightening, they are reset to ∞. With this in mind, the high-level flow of propagating our example update is as follows. When the time points in the root clique ι are tagged, agents B and H update their involved constraints to new values: $G_E^B - I_E^H \in [25, 75]$ and $I_E^H - z \in$ [2:15, 3:20]. When clique ι is done, agent H propagates the change to agents A and B, the respective owners of ϵ and ζ. Note that the clique tree now decomposes into two independent parts where propagation continues simultaneously. When DIPPC finds that a change cannot or need not be propagated further (either because the current clique tree node is a leaf or the propagation causes no changes in the clique), it sends a PROP-DONE notification back up to that clique node's parent. This parent node in turn tells *its* parent that it is done when all its children have indicated they are. Thus, propagation is complete when a PROP-DONE notification reaches the root from all its children—in this case, when it reaches clique ι from ϵ and ζ.

Continuing the example propagation, B immediately returns a PROP-DONE notification for ζ, whereas A tags D_E^A, the remaining time point in ϵ, causing A and H to update their inter-agent constraint to $D_E^A - I_E^H \in [45, 100]$. Next, the owners of α, β, δ and θ are sent a PROP message, but propagation is required only for θ (by agent H). This leads to the final edge update: $I_S^H - z \in$ [1:35, 2:50]. All PROP-DONE notifications bubble upwards to the root ι, after which agent H concludes that propagation is complete.

Algorithm 2: DIPPC

Input: Edge $\{a, b\}$ with new weight $w'_{a\to b}$

if $w'_{a\to b} \geq w_{a\to b}$ **then return** CONSISTENT
if $w'_{a\to b} + w_{b\to a} < 0$ **then return** INCONSISTENT
MakeLive$(a, 0, \infty)$
MakeLive$(b, \infty, 0)$
await LIVE-DONE for all LIVE sent
$C \leftarrow$ FindCommonClique(a, b)
send PROP $(C, \{a, b\})$ to $Owner(C)$
await PROP-DONE for PROP $(C, \{a, b\})$

Before going into more implementation details of DIPPC, we first discuss its relative strengths and weaknesses. We start with the strengths it has in common with the DI△STP algorithm. Like its counterpart, DIPPC exploits the natural load-balancing of cliques among agents that results from distributed triangulation by DP^{3}C . The privacy properties of DP^{3}C also extend to DIPPC and guarantee that if an update is local in scope, it is processed independently of all other agents. However, in contrast to DI△STP, a major disadvantage of the DIPPC algorithm is that it is not as asynchronous. Thus, the level of concurrency that DIPPC achieves is subject to how quickly the clique tree structure branches across multiple agents. The upside of this increased synchronicity is that by visiting nodes using a simplicial construction ordering like the IPPC algorithm, DIPPC will visit any given edge at most once, minimizing the total effort of the system.

Algorithmic Details. Apart from the top-level message PROP and notification PROP-DONE, the algorithm requires two additional message-notification pairs forming a middle and a lower layer. When a clique is activated, the agent responsible for that clique sends a TAG message to (the owners of) new vertices in the clique, i.e., vertices that were not present in any previously activated clique. This corresponds exactly to the original IPPC algorithm: when a vertex v is tagged, its owner can efficiently determine whether it is live or dead: whether any edges incident on v must be changed or not. When v is found to be live, its owner communicates this to all agents connected to v by an external edge using a message of type LIVE, which includes the distances $D_a^{\downarrow}[v]$ and $D_b^{\uparrow}[v]$, like in the original IPPC algorithm. The LIVE message, upon receipt, is immediately acknowledged with a LIVE-DONE notification. Finally, once an agent has received these notifications for all LIVE messages it has sent, it informs the owner of the active clique with a TAG-DONE notification that it is done. When all TAG messages have been responded to in this fashion, propagation continues using PROP messages to the clique node's children in the tree.

Note that in all pseudocode, *waiting* for some number of notifications to arrive does not mean that the process does nothing at all. Instead, a counter is decremented every time

a notification of the appropriate type arrives while the agent continues receiving and responding to messages and notifications of other types. As soon as the counter reaches zero, operation of the procedure continues as described.

The MakeLive procedure, in short, iterates over each neighbor v of a newly-live vertex u, and either updates the edge if v is live or updates the distance values for v otherwise. It also informs other agents that know about u that it is now live, and maintains a *live counter*. This counter is used to keep track of the number of live vertices in a clique. When a clique is activated but contains fewer than two live vertices, propagation immediately stops. Once again, a similar provision was present in the original IPPC algorithm.

Procedure HandleMsg

Input: Incoming message m

switch type of m
- **case** LIVE $(u, d_{u\to a}, d_{b\to u})$
  - MakeLive$(u, d_{u\to a}, d_{b\to u})$
  - **send** LIVE-DONE (u) to $Owner(u)$
- **case** PROP $(C_{\text{cur}}, C_{\text{old}})$
  - **if** $LiveCount[C_{\text{cur}}] \geq 2$ **then**
    - **forall** $u \in C_{\text{cur}} \setminus C_{\text{old}}$ **do**
      - **send** TAG (u) to $Owner(u)$
    - await TAG-DONE for all TAG sent
    - **forall** $C' \in Adj(C_{\text{cur}}) \setminus \{C_{\text{old}}\}$ **do**
      - **send** PROP (C', C_{cur}) to $Owner(C')$
    - await PROP-DONE for all ACTIVATE sent
  - **send** PROP-DONE to $Owner(C_{\text{old}})$
- **case** TAG $(ToTag)$
  - **forall** $u \in ToTag$ **do**
    - set state of u to tagged
    - **forall** $\{u, v\} \in pending(u)$ **do**
      - $w_{u\to v}$.update$(D_a^{\downarrow}[u] + w_{a\to b} + D_b^{\uparrow}[v])$
      - $w_{v\to u}$.update$(D_a^{\downarrow}[v] + w_{a\to b} + D_b^{\uparrow}[u])$
    - **if** no changes **then** set state of u to dead
  - **forall** vertices $u \in ToTag$ not marked dead **do**
    - MakeLive$(u, D_a^{\downarrow}[u], D_b^{\uparrow}[u])$
  - await LIVE-DONE for all LIVE sent
  - **send** TAG-DONE to originator of TAG message

Empirical Evaluation

We compare the performance of both new distributed algorithms with the state-of-the-art centralized approach.

Experimental Setup. Our problems[3] come from two sources. The first is the *BDH problem set*, which uses Boerkoel and Durfee's (2011) multiagent adaptation of Hunsberger's (2002) original random STP generator. Each MaSTN has N agents each with start time points and end time points for 10 activities, which are subject to various duration, makespan, and other local constraints. In addition, each MaSTN has X external contraints. We evaluate algorithms

Procedure MakeLive$(u, d_{u\to a}, d_{b\to u})$

Input: Vertex u, distances $d_{u\to a}$ and $d_{b\to u}$

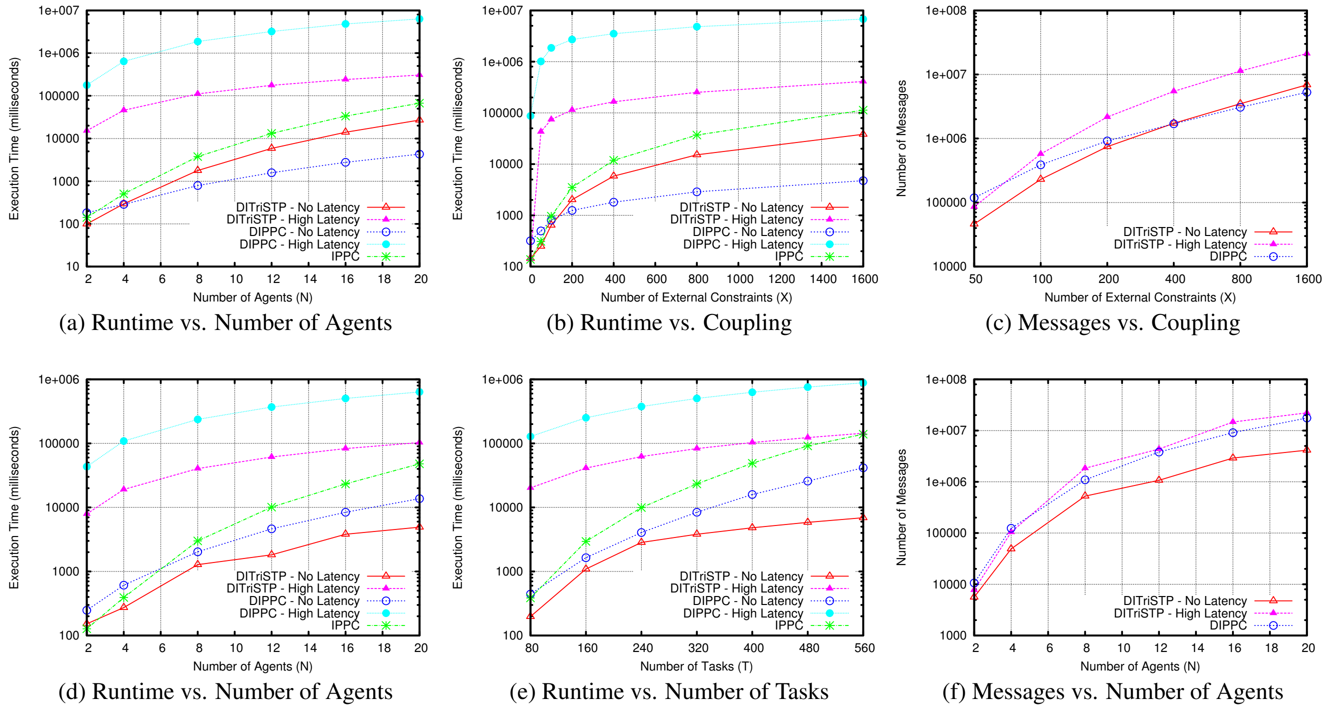
set state of u to live
increment $LiveCount[u]$
$D_a^{\downarrow}[u] \leftarrow d_{u\to a}$
$D_b^{\uparrow}[u] \leftarrow d_{b\to u}$
forall $v \in V$ **such that** $\{u, v\} \in E$ **do**
- **if** v has not yet been tagged **then**
  - $D_a^{\downarrow}[v]$.update$(w_{v\to u} + d_{u\to a})$
  - $D_b^{\uparrow}[v]$.update$(d_{b\to u} + w_{u\to v})$
  - **if** v is mine **then** $pending(v).append(\{v, u\})$
- **else**
  - $w_{u\to v}$.update$(D_a^{\downarrow}[u] + w_{a\to b} + D_b^{\uparrow}[v])$
  - $w_{v\to u}$.update$(D_a^{\downarrow}[v] + w_{a\to b} + D_b^{\uparrow}[u])$
- **if** $v \prec u$ **then** increment $LiveCount[v]$
- **if** u is mine but v is not, and $Owner(v)$ has not yet been informed **then**
  - **send** LIVE $(u, d_{u\to a}, d_{b\to u})$ to $Owner(v)$

both on how well they *scale* in response to an increasing number of agents ($N \in \{2, 4, 8, 12, 16, 20\}, X = 50 \cdot (N - 1)$) and also on how they perform across various degrees of agent *coupling* ($N = 16, X \in \{0, 50, 100, 200, 400, 800, 1600\}$).

The second source of problems is the *WS problem set* derived from a multiagent factory scheduling domain (Wilcox and Shah 2012). These randomly generated MaSTNs simulate N agents working together to complete T tasks in the construction of a large structural workpiece in a manufacturing environment using realistic task duration, wait, and deadline constraint settings. This emulates a factory manager who uses domain knowledge to progressively refine the space of schedules until only feasible schedules remain. Like before, we evaluate algorithms both as number of agents increases ($N \in \{2, 4, 8, 12, 16, 20\}, T = 20 \cdot N$) and also as the total number of tasks increases ($N = 16, T \in \{80, 160, 240, 320, 400, 480, 560\}$).

In our evaluations, we only consider consistent MaSTNs (i.e., no overconstrained networks). Constraints are divided into two sets: (i) structural constraints, where bounds are relaxed to their least constraining possible settings; and (ii) refinement constraints, representing the true underlying constraint bound values. PPC is established over the set of structural constraints which then acts as the initial input to our incremental algorithms. Then, refinement constraints are randomly chosen (with uniform probability) and fed into the network, one at a time, until all constraints have been incorporated. We wait until each update is fully processed and quiescence is reached before feeding in the subsequent constraint. The temporal reference point (i.e., z) is special in the sense that it is not unique to any particular agent but simultaneously known by all agents. We emulate this with a

[3]Problem sets available at:
http://dx.doi.org/10.4121/uuid:3e6a8869-8500-4979-bcfa-361e07fc0dc6



(a) Runtime vs. Number of Agents (b) Runtime vs. Coupling (c) Messages vs. Coupling

(d) Runtime vs. Number of Agents (e) Runtime vs. Number of Tasks (f) Messages vs. Number of Agents

Figure 4: Results for the BDH (top) and WS (bottom) problem sets.

special reference 'agent' to and from which there is no cost for sending or processing messages (so agents have zero-cost access to a synchronized clock).

We simulate a multiagent system on a 2.4 GHz processor with 4 GB of memory using a message-driven approach.[4] Our simulator systematically shares the processor among agents, tracking the wall-clock time each agent spends executing its code. To ensure that messages are delivered in the right order, we maintain a priority queue of pending messages ordered by their timestamp. Within the simulated multiagent environment, agents can either idly wait for incoming messages or check in a non-blocking way whether messages are pending. The simulation ends once all agents are idle and there are no pending messages. In our case the simulation is kicked off by a special "nature" agent, which posts refinement constraints one by one by sending a message to one of the agents involved, and waits for quiescence every time. Our setup allows us to simulate message latency by adding a delay to the timestamp of a message before inserting it into the queue. In our experiments, we penalized each message with a delay chosen with uniform probability from a range $[0, d_{\max}]$, where $d_{\max}$ is set to 0 or 100 milliseconds to emulate *No* and *High* latency situations respectively. For each parameter setting, we report the mean over 50 unique problem instances.

Empirical Comparison. Our experiments are aimed at discovering which algorithm performs best under which circumstances. To do this, we compare our two new distributed algorithms, in both high and no latency settings, against each other and against IPPC, the state-of-the-art centralized approach. To improve clarity, we omit including the centralized version of DI△STP since IPPC outperformed it by a steady, nearly order-of-magnitude factor in expectation. Our experiments also implicitly validate that constraints can be incrementally propagated on distributed, MaSTNs without requiring additional centralization. Figure 4 displays a comparison of our distributed incremental algorithms where we evaluate both (simulated) algorithm runtimes and the number of messages passed. Note that these runtimes reflect the total (simulated wall-clock) time elapsed, not the summed computational effort.

We start by describing the run-time results from our BDH problem set, in Figures 4a–b. With no message latency, both DI△STP and DIPPC achieve reduced execution time compared to IPPC, with DIPPC improving by up to an order of magnitude as the number of agents and external edges grow. Even though DI△STP underachieved compared to DIPPC with no message latency, it must be noted that it achieved similarly impressive speedups over its centralized counterpart. This demonstrates that when there is no message latency, both algorithms are able to effectively load-balance their efforts. At high latency, DI△STP exhibits a steady order-of-magnitude improvement over DIPPC. For high message latency, neither distributed algorithm outperforms IPPC, which suggests that there are be cases where centralization is most computationally efficient. Note however that IPPC's runtime increases faster, indicating there may eventually be a

[4]Java multiagent simulator implementation available at:
http://dx.doi.org/10.4121/uuid:d68d75a0-ede1-4b0c-b298-d2181a7c6331

cross-over point for problems with sufficiently many agents and external constraints where our algorithms would outperform IPPC, even at high latency.

As shown in Figure 4c, when there is no message latency, both new algorithms send similar numbers of messages. Regardless of latency, DIPPC will—by design—always send the same number of messages in the same order, whereas latency increases the number of redundantly processed triangles by DI△STP and consequently increases the number of (likewise redundant) messages. However, the extra messages sent by DI△STP propagate information through the network faster, and while redundant computation is performed, the chances that DI△STP can complete sooner than the more synchronous DIPPC also improve.

The results from our WS problem set, displayed in Figures 4d–f, are very similar in nature to those from our BDH problems. We briefly highlight a few key differences. First, when there is no latency, DI△STP outperforms DIPPC, which both outperform IPPC. We conjecture that the gains made by the DI△STP are due to the more realistically structured problems of the WS set. An update in a WS instance is more likely to cause more and longer paths of propagation than an update in the more random structure of a BDH instance. In such cases, DI△STP is better suited to short-circuit long paths of propagation (albeit occasionally prematurely), as compared to DIPPC, which carefully synchronizes path traversal to avoid any wasted effort. A second difference of note is that, at high latency, the prospect of a cross-over between the IPPC curve and the curves of our distributed algorithms is more evident. This indicates that realistic, well-structured MaSTNs may have more to gain from the distribution of temporal network management.

The run-time performances of both algorithms are directly impacted by the density of the resulting chordal graph. This can be seen in Figure 4b, where runtime of both algorithms increases as the number of external constraints increases. The actual density of the chordal STNs varies from 2.0% to 45%, where 0% and 100% respectively represent a graph without any edges and a complete graph. In general, correlation of density is negative with the number of agents N and with the number of tasks T in the WS problem set, whereas it is positive with the number of external constraints X in the BDH problem set.

In addition to the results shown in these figures, we also empirically verified our hypothesis that DI△STP does a better job at minimizing agent idleness while DIPPC minimizes the total amount of work overall. We ground this phenomenon with a result from the BDH problem set; similar trends hold for the WS set. With $N = 16$, $X = 1600$, and no latency, DIPPC executes 8 times faster, does 5 times less work (sum of agents' execution times), and achieves 37% higher agent utilization (portion of time not spent idling) than DI△STP. For the same problems with high latency, the total amount of work performed is stable for DIPPC, while its advantage over DI△STP grows to a total factor of 7. However, the latter's agent utilization is now over 100 times higher than DIPPC, whose total execution now takes 16 times longer than DI△STP. Boerkoel and Durfee (2010) report that the speedup of the DP^{3}C algorithm, which must propagate all edges in the MaSTN, decreases as the number of external constraints increases. Interestingly, in the incremental setting, which only needs to propagate the impact of an updated edge, we found the opposite to be true: an increase in the number of external constraints *increased* the opportunities for concurrency by branching propagation to more agents.

In short, the meticulousness of DIPPC to avoid any superfluous computation makes it ideal for situations with low or no message latency (e.g., parallel systems) while the asynchronous nature of DI△STP makes it better suited to handle scenarios with high message latency (e.g., messages that must travel the Internet) or with long, structured propagation paths. In many realistic scenarios, agents may interleave managing their temporal networks with, e.g., looking for improved plans or new scheduling opportunities (Barbulescu et al. 2010). Here, concerns about high message latency are mitigated: DIPPC's idle time may be put to good use by granting agents increased time for managing other important tasks. For example, an agent could spend its extra time evaluating 'what-if' scenarios on a copy of its local network or by tracking and rolling back changes, without global commitment. In other settings, constraints may arise more quickly than agents are able to process them. DI△STP implicitly handles the asynchronous arrival of constraint updates and may here have an advantage over DIPPC, which must wait until each update is processed to completion.

Conclusion

Distributed maintenance of temporal networks is crucial to the coordination of multiagent systems, allowing agents to achieve increased autonomy and privacy over their local schedules. We proposed two new *distributed* algorithms for *incrementally* maintaining PPC on distributed, MaSTNs without requiring additional centralization: (i) DI△STP, which allows for the fast, asynchronous propagation of updates throughout the network; and (ii) DIPPC, which carefully propagates updates through a clique tree representation of the network, thus meticulously avoiding redundant effort. We demonstrated empirically that when message latency is minimal, both algorithms achieve reduced solve times—by upwards of an order of magnitude—as compared to the state-of-the-art centralized approach, especially as problems grow in the number of agents or external constraints. However, as message latency increases, the relative performance of DI△STP improves due to its asynchronous nature. In the future, we would like to investigate a hybrid approach that balances the benefits of asynchronicity with the advantages of eliminating redundant behavior. One possibility is a variant of DI△STP that instead maintains triangles in a priority queue ordered by the clique tree distances. Another is modifying DIPPC to eliminate some synchronization, thus increasing agents' ability to perform anticipatory computation.

Acknowledgments

We thank the anonymous reviewers for their suggestions. This work was supported, in part, by the NSF under grant IIS-0964512, by Boeing Research and Technology, and by a University of Michigan Rackham Fellowship.

References

Barbulescu, L.; Rubinstein, Z. B.; Smith, S. F.; and Zimmerman, T. L. 2010. Distributed Coordination of Mobile Agent Teams. In *Proc. of AAMAS-10*, 1331–1338.

Bliek, C., and Sam-Haroud, D. 1999. Path Consistency on Triangulated Constraint Graphs. In *Proc. of IJCAI-99*, 456–461.

Boerkoel, J. C., and Durfee, E. H. 2010. A Comparison of Algorithms for Solving the Multiagent Simple Temporal Problem. In *Proc. of ICAPS-10*, 26–33.

Boerkoel, J. C., and Durfee, E. H. 2011. Distributed Algorithms for Solving the Multiagent Temporal Decoupling Problem. In *Proc. of AAMAS 2011*, 141–148.

Boerkoel, J. C., and Durfee, E. H. 2013. Distributed Reasoning for Multiagent Simple Temporal Problems. *Journal of Artificial Intelligence Research (JAIR),* To Appear.

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 COPLAS Workshop at ICAPS'08*, 17–24.

Castillo, L.; Fernández-Olivares, J.; García-Pérez, O.; and Palao, F. 2006. Efficiently Handling Temporal Knowledge in an HTN Planner. In *Proc. of ICAPS-06*, 63–72.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. In *Knowledge representation*, volume 49, 61–95. The MIT Press.

Hunsberger, L. 2002. Algorithms for a Temporal Decoupling Problem in Multiagent Planning. In *Proc of AAAI-02*, 468–475.

Laborie, P., and Ghallab, M. 1995. Planning with Sharable Resource Constraints. In *Proc. of IJCAI-95*, 1643–1649.

Planken, L. R.; De Weerdt, M. M.; and Van der Krogt, R. P. J. 2008. P3C: A New Algorithm for the Simple Temporal Problem. In *Proc. of ICAPS-08*, 256–263.

Planken, L. R.; De Weerdt, M. M.; and Van der Krogt, R. P. J. 2011. Computing All-Pairs Shortest Paths by Leveraging Low Treewidth. In *Proc. of ICAPS-11*, 170–177.

Planken, L. R.; De Weerdt, M. M.; and Yorke-Smith, N. 2010. Incrementally Solving STNs by Enforcing Partial Path Consistency. In *Proc. of ICAPS-10*, 129–136.

Shah, J. A., and Williams, B. C. 2007. A fast incremental algorithm for maintaining dispatchability of partially controllable plans. In *Proc. of ICAPS-07*, 296–303.

Tarjan, R. E., and Yannakakis, M. 1984. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing* 13(3):566–579.

Wilcox, R. J., and Shah, J. A. 2012. Optimization of Multi-Agent Workflow for Human-Robot Collaboration in Assembly Manufacturing. In *Proc. of AIAA Infotech@Aerospace*.

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.