Efficient Computation of Minimal Point Algebra Constraints by Metagraph Closure

Alfonso Gerevini and Alessandro Saetti
Dipartimento di Elettronica per l'Automazione
Università degli Studi di Brescia, via Branze 38, 25123 Brescia, Italy
{gerevini,saetti}@ing.unibs.it

Abstract. Computing the minimal network (or minimal CSP) representation of a given set of constraints over the Point Algebra (PA) is a fundamental reasoning problem. In this paper we propose a new approach to solving this task which exploits the timegraph representation of a CSP over PA. A timegraph is a graph partitioned into a set of chains on which the search is supported by a metagraph data structure. We introduce a new algorithm that, by making a particular closure of the metagraph, extends the timegraph with information that supports the derivation of the strongest implied constraint between any pair of point variables in constant time. The extended timegraph can be used as a representation of the minimal CSP. We also compare our method and known techniques for computing minimal CSPs over PA. For CSPs that are sparse or exhibit chain structure, our approach has a better worst-case time complexity. Moreover, an experimental analysis indicates that the performance improvements of our approach are practically very significant. This is the case especially for CSPs with a chain structure, but also for randomly generated (both sparse and dense) CSPs.

1 Introduction

Constraint-based qualitative temporal reasoning is a widely studied area with application to various fields of AI (for a recent survey see [2]). The Point Algebra [9, 10] is one of the first and most prominent frameworks for representing qualitative temporal constraints and reasoning about them. PA consists of three basic relations between time point variables that are jointly exhaustive and pairwise disjoint ($<$, $>$, $=$), all possible unions of them ($\leq$, $\geq$, $\neq$, and $\top$, where $\top$ is the universal relation), and of the empty relation. The *convex* Point Algebra contains all the relations of PA except $\neq$.

Given a set C of temporal constraints over PA (or a temporal CSP), a fundamental reasoning problem is computing the minimal CSP representation of C.[1] A temporal CSP is *minimal* if, for every pair of variables i, j, the relation R between i and j is the strongest relation (or minimal constraint) between i and j that is entailed by the CSP. In other words, every basic relation $r \in R$ is feasible, i.e., there exists a solution of the temporal CSP where the values assigned to i and j satisfy r.

[1] This problem is also called *deductive closure* problem in [9], *minimal labeling* problem in [5, 7] and computing the *feasible relations* in [6].

Computing the minimal CSP of a temporal CSP involving n variables can be accomplished in $O(n^3)$ time by using a path-consistency algorithm, if the CSP is over the convex PA [9, 10], and in $O(n^3 + n^2 \cdot c_{\neq})$, if the CSP is over the full PA [6, 4], where $c_{\neq}$ is the number of input $\neq$-constraints.

An alternative approach to qualitative temporal reasoning in the context of PA is the "graph-based approach" [1, 3]. Instead of computing the minimal CSP, we build a particular graph-based representation of the input CSP that supports efficient computation of the minimal constraints at query time. This method has been proposed with the aim of addressing scalability especially for large data sets forming sparse CSPs (in which the number of constraints is less than quadratic) and exhibiting particular structure (e.g., a collection of time chains [3] or series-parallel graphs [1]).

The minimal CSP representation supports the derivation of the strongest entailed relation between any pair of variables in constant time. On the contrary, in the graph-based representation this has a computational cost that depends on the structure and sparseness of the input temporal CSP; in the best case it can be constant time, while in the worst case it can be quadratic time with respect to the number of the CSP point variables. On the other hand, in practice computing the minimal CSP representation using known techniques is significantly slower than computing the graph-based representation of a sparse CSP [1, 3].

In this paper, we investigate a combined approach. We propose a new method for computing the minimal CSP representation by exploiting the *timegraph* representation [3]. A timegraph is a graph partitioned into a set of time chains on which the search is supported by a *metagraph* data structure. We introduce a new algorithm that, by making a particular closure of the metagraph, extends the timegraph with information that supports the derivation of the strongest entailed relation between any pair of point variables in constant time. The extended timegraph can be seen as a representation of the minimal CSP. By using our approach the (explicit) minimal CSP can be computed in $O(\hat{n} \cdot \hat{e} + n^2)$ time (convex PA) and $O(\hat{e} \cdot \hat{n} + \hat{e}_{\neq} \cdot (\hat{e} + \hat{n}) + n^2)$ time (full PA), where $\hat{n}$, $\hat{e}$ and $\hat{e}_{\neq}$ are the metanodes, metaedges and $\neq$-metaedges, respectively, in the timegraph.

We compare our method and the known techniques for computing minimal CSPs over PA. For CSPs that are sparse or exhibit chain structure, our approach has a better worst-case time complexity. Moreover, an experimental analysis indicates that in practice our approach is significantly faster. This is the case not only for CSPs with a chain structure, but also for general randomly generated (both sparse and dense) CSPs.

The paper is organized as follows. Section 2 gives the necessary background. Section 3 presents our new method for computing the minimal CSP. Section 4 concerns the experimental analysis. Finally, Section 5 gives the conclusions.

2 Background: TL-graphs, Timegraphs and Metagraphs

A **temporally labeled graph** (*TL-graph*) [3] is a graph with at least one vertex and a set of labeled edges, where each edge (v, l, w) connects a pair of distinct

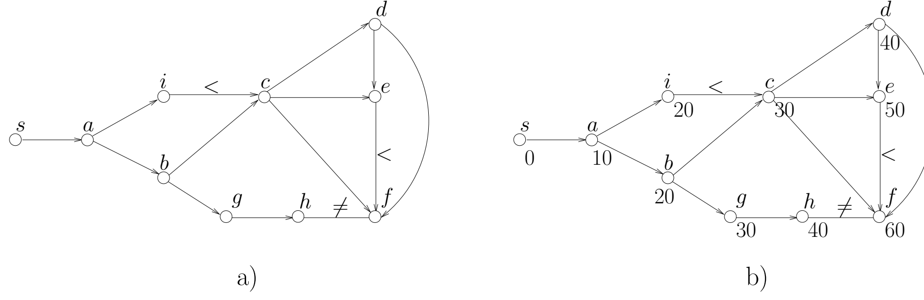

Fig. 1. Examples of TL-graph and ranked TL-graph. Edges with no label are assumed to be labeled "$\leq$".

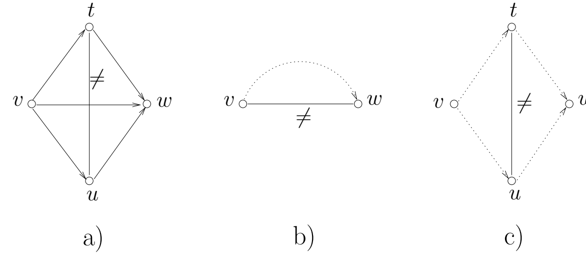

Fig. 2. a) van Beek's forbidden graph; b) and c) the two kinds of implicit $<$ relation in a TL-graph. Edges with no labels are assumed to be labeled "$\leq$". Dotted arrows indicate paths, solid lines $\neq$-edges. In each graph there is an implicit $<$ between v and w.

vertices v, w representing point variables. Each edge represents a PA-constraint and is either directed and labeled $\leq$ or $<$, or undirected and labeled $\neq$. Figure 1.a) shows an example of TL-graph. A path in a TL-graph is called a $\leq$-path if each edge on the path has label $<$ or $\leq$. A $\leq$-path is called a $<$-path if at least one of the edges has label $<$.

A TL-graph $\mathcal{G}$ contains an *implicit* $<$ relation between two vertices v_1, v_2 when the strongest relation entailed by the temporal CSP from which $\mathcal{G}$ has been built is $v_1 < v_2$ and there is no $<$-path from v_1 to v_2 in $\mathcal{G}$. Figures 2.b) and 2.c) show the two possible TL-graphs which give rise to an implicit $<$ relation. All TL-graphs with an implicit $<$ relation contain one of these subgraphs [4, 6].

An acyclic TL-graph without implicit $<$ relations is an *explicit* TL-graph. In order to make explicit a TL-graph containing implicit $<$ relations, we can add new edges with label $<$ [3]. For example, in Figure 2 we add the edge $(v, <, w)$ to the graph. An important property of an explicit TL-graph is that it entails $v \leq w$ if and only if there is a $\leq$-path from v to w; it entails $v < w$ if and only if there is a $<$-path from v to w, and it entails $v \neq w$ if and only if there is a $<$-path from v to w or from w to v, or there is an edge $(v, \neq, w)$.

Given a CSP C over PA with c constraints, we can construct a TL-graph $\mathcal{G}$ representing C in $O(c)$ time; In order to check consistency of C and transform $\mathcal{G}$ into an equivalent acyclic TL-graph, we can use van Beek's method for PA [6].

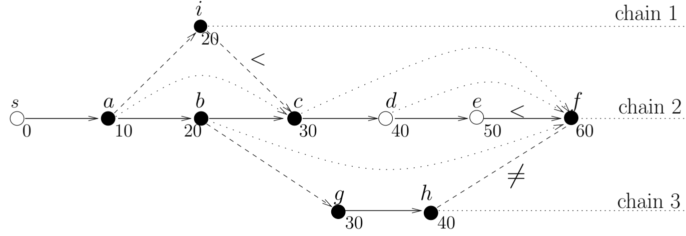

Fig. 3. The timegraph of the TL-graph of Figure 1, with transitive edges and auxiliary edges omitted. Dotted links between nodes on the same chain are nextgreater links. Edges with no label are assumed to be labeled "$\leq$".

If the TL-graph (C) is consistent, each SCC is collapsed into an arbitrary vertex v within that component, and all the cross-component edges entering or leaving the component are appropriately transferred to v.

A *ranked TL-graph* [3] is a simple but powerful extension of an acyclic TL-graph. In a ranked TL-graph (see Figure 1.b) , each vertex (time point) has a *rank* associated with it. The rank of a vertex v can be defined as the length of the longest $\leq$-paths to v from a source vertex s of the TL-graph representing the "universal start time", times a rank distance increment k [3]. The special vertex s has no predecessor and its successors are all the vertices of the graph that have no other predecessor. As observed in [1, 3], the use of the ranks can significantly speed up the search for a path from a vertex p to another vertex q: the search can be pruned whenever a vertex with a rank greater than or equal to the rank of q is reached.

A **timegraph** [3] is an acyclic ranked TL-graph partitioned into a set of *time chains*, such that each vertex is on one and only one time chain. A time chain is a $\leq$-path, plus possibly "transitive edges" connecting pairs of vertices on the $\leq$-path. Distinct chains of a timegraph can be connected by *cross-chain edges*. Vertices connected by cross-chain edges are called *metanodes*. Cross-chain edges and certain auxiliary edges connecting metanodes on the same chain are called *metaedges*. The auxiliary edges connect each metanode to the first (last) successor (predecessor) metanode on the same chain with an outgoing cross-edge, and to the first (last) successor (predecessor) metanode on the same chain with an incoming cross-edge. These auxiliary edges are called the *NextOut* (*PrevOut*) and *NextIn* (*PrevIn*) edges of a metanode. The metanodes and metaedges of a timegraph T form the **metagraph** of T.[2]

Figure 3 shows the timegraph built from the TL-graph of Figure 1. All vertices except d, e and s are metanodes. The edges connecting vertices a to i, i to c,

[2] The purpose of the metagraph is not to be an autonomous structure independent of the rest of the graph, but a support structure to facilitate the graph search conducted during the construction of the timegraph data structures or at query time. Our implementation of the timegraph algorithms does handle the potential "pathological case" identified in [1]. This is done by considering as the successors of a metanode v on a chain c *all* successors of v on c with at least one outgoing cross-chain metaedge.

b to g, h with f, are metaedges. Dotted edges are special links called *nextgreaters* that are computed during the construction of the timegraph and that indicate for each vertex v the nearest descendant v' of v on the same chain as v such that the represented CSP entails $v < v'$.

In a timegraph the main purpose of the ranks is to support the computation of the strongest entailed relation between two vertices on the same chain in constant time: given two vertices v_1 and v_2 on the same chain such that the rank of v_2 is greater than the rank of v_1, if the rank of the nextgreater of v_1 is less than or equal to the rank of v_2, then the timegraph entails $v_1 < v_2$, otherwise it entails $v_1 \leq v_2$. For example, the timegraph of Figure 3 entails $a < d$ because a and d are on the same chain, and the rank of the nextgreater of a is less than the rank of d.

Given a CSP C over PA, in order to build a timegraph representation of C, we start from a TL-graph $\mathcal{G}$ representing C. The construction of the timegraph from $\mathcal{G}$ consists of four main steps: checking the consistency of $\mathcal{G}$ (and C), ranking of the graph vertices, formation of the time chains and the metagraph, and making explicit the implicit $<$ relations. The total time complexity of building the timegraph for a CSP C consisting of c PA-constraints involving n point variables is [3]:

- $O(n + c + \hat{e} \cdot \hat{n})$, if C is over the convex PA,
- $O(n + c + \hat{e} \cdot \hat{n} + \hat{e}_{\neq} \cdot (\hat{e} + \hat{n}))$, if C is over the full PA,

where $\hat{n}$ is the number of metanodes in the timegraph, $\hat{e}$ is the number of metaedges and $\hat{e}_{\neq}$ is the number of metaedges labeled $\neq$. It is worth noting that typically $\hat{n}$ is smaller than n, $\hat{e}$ is smaller than c, and $\hat{e}_{\neq}$ is smaller than then the number of input $\neq$-constraints.

Concerning querying the strongest entailed relations between two point variables p_1 and p_2 represented in a timegraph, there are four cases in which this can be accomplished in constant time: (1) p_1 and p_2 are alternative names of the same vertex (the strongest entailed relation is "="); (2) the vertices v_1 and v_2 corresponding to p_1 and p_2 are on the same time chain; (3) v_1 and v_2 are not on the same chain and have the same rank, and there is no $\neq$ edge between them (the strongest entailed relation is "$\top$"); (4) v_1 and v_2 are connected by a $\neq$-edge (the strongest entailed relation is $\neq$). In the remaining cases an explicit search on the metagraph needs to be performed. If there exists at least one $<$-path from v_1 to v_2, then the answer is $v_1 < v_2$. If there are only $\leq$-paths (but no $<$-paths) from v_1 to v_2, then the answer is $v_1 \leq v_2$ [3]. (Analogously for the paths from v_2 to v_1.) Such a graph search can be accomplished in $O(\hat{e} + \hat{n})$ time.

3 Chain Closure for Timegraphs

In this section we extend the timegraph representation with additional information that can be exploited to compute the strongest entailed relation (or minimal constraint) between *any* pair of variables in constant time. This information consists of two additional links for each metanode v with an outgoing cross-chain edge and each chain c:

– *ChainLess*(v, c), which is the first node w on c such that $v < w$ is entailed by the timegraph (there is a $<$-path from v to w);
– *ChainLeq*(v, c), which is the first node t on c such that $v \leq t$ is entailed by the timegraph (there is a $\leq$-path from v to t).

When one of these links is undefined, the corresponding link has value "*null*". We call the set of *ChainLess* and *ChainLeq* links the **chain closure** (or *metagraph closure*) of the timegraph.

3.1 Chain Closure Algorithm

The algorithm in Figure 4, ChainClosure, consists of three nested loops. The most external loop considers each chain c; the second nested loop considers each node v with outgoing cross-chain edge on c as a start node for a search; the third nested loop performs a complete search on the metagraph for computing the *ChainLess* and *ChainLeq* links for v and each time chain.

The first search starting from a metanode on a chain c starts from the last metanode $v = LastOut(v)$ on c with an outgoing cross-edge (step 3). Once the algorithm has found all paths from v to the metanodes on each chain $c' \neq c$, the *ChainLess* and *ChainLeq* links of v for all these chains have been computed, and the algorithm "moves back" to the previous metanode on c, $v = PrevOut(v)$, with an outgoing cross-chain edge (step 23) to initiate another search (step 5).

The search from v is conducted by maintaining a frontier of nodes to be visited (*Open*). Each time the search visits a metanode s *ChainLess*$(v, Chain(s))$ and *ChainLeq*$(v, Chain(s))$ are updated, where $Chain(s)$ denotes the chain of s (steps 7–14). Since the order in which the chains are processed by the external loop corresponds to their increasing number, if $Chain(s) < Chain(v)$, then, for all metanodes on $Chain(s)$, the *ChainLess* and *ChainLeq* links of s have already been computed, and so they can be used to update the *ChainLess* and *ChainLeq* of v (steps 7-8). For example, if $Rank(ChainLess(s, c')) < Rank(ChainLess(v, c'))$, where $c \neq c'$, then *ChainLess*(v, c') is set to *ChainLess*(s, c'), and similarly for *ChainLeq*(v, c').[3] A similar updating is done when $Chain(s) = Chain(v)$ and s has an outgoing cross-chain edge; this propagates to v the *ChainLess* and *ChainLeq* information previously computed for s. Steps 9–14 consider all the other cases in which the *ChainLess* and *ChainLeq* links of v for $Chain(s)$ need to be updated. The flag $Rel(s) \in \{<, \leq, null\}$ keeps track of the strongest path (relation) from v to s found by the search (*null* means "no path found" so far).

After visiting s, the frontier *Open* is extended with the child nodes of s (steps 15–20), provided that they satisfy certain conditions illustrated below. The purpose of these conditions is pruning the successor nodes for which we can anticipate that the current *ChainLess* and *ChainLeq* of v cannot be refined by continuing the search from them. The search from v terminates when $s = Pop(Open)$ is *null* (step 6), i.e., when *Open* becomes empty.

In order to make the search more efficient, not all child nodes of the current node s are added to *Open*. This pruning is performed by exploiting the chain

[3] We omit the details of this updating process, which is straightforward.

Algorithm: ChainClosure

Input: a timegraph T;
Output: T extended with chain closure (*ChainLeq* and *ChainLess* links);

1. $Open :=$ empty list;
2. **for each** chain c (processed according to their increasing number) **do**
3. $v := LastOut(c)$; $Visited :=$ empty list;
4. **while** $v \neq null$
5. $s := v$; add s to *Visited*;
6. **while** $s \neq null$
7. **if** $Chain(s) \leq Chain(v)$ and $s \neq v$ and s has outgoing metaedges **then**
8. Update $ChainLeq(v, c')$, $ChainLess(v, c')$ with $ChainLeq(s, c')$ and $ClainLess(s, c')$, respectively, for each chain $c' \neq c$;
9. **if** $Chain(v) \neq Chain(s)$ and $Rel(s) = <$ **then**
10. **if** $Rank(ChainLess(v, Chain(s))) > Rank(s)$ **then** $ChainLess(v, Chain(s)) := s$;
11. **if** $ChainLeq(v, Chain(s)) \neq null$ and $Rank(ChainLeq(v, Chain(s))) \geq Rank(s)$ **then** $ChainLeq(v, Chain(s)) := null$;
12. **if** $Chain(v) \neq Chain(s)$ and $Rel(s) = \leq$ **then**
13. **if** $Rank(ChainLess(v, Chain(s))) > Rank(s)$ and $Rank(ChainLeq(v, Chain(s))) > Rank(s)$ **then** $ChainLeq(v, Chain(s)) := s$;
14. **if** $NG(s) \neq null$ and $Rank(ChainLess(v, Chain(s)) > Rank(NG(s))$ **then** $ChainLess(v, Chain(s)) := NG(s)$;
15. **if** $Chain(s) > Chain(v)$ or $s = v$ or s has no outgoing cross-chain edges **then**
16. $S :=$ set of metanodes to which s is connected by outgoing cross-chain edges
17. **if** $s = v$ or $RelChain(s) = null$ or ($RelChain(s) = \leq$ and $Rel(s) = <$) **then** $S := S \cup \{s' \mid s'$ is successor of s on $Chain(s)$ through the *NextOut* links$\}$
18. **for each** $w \in S$ **do**
19. **if** $Rel(s) = <$ or $(s, <, w) \in \hat{E}$ or ($Chain(s) = Chain(w)$ and $s < w$) **then** $update := <$ **else** $update := \leq$;
20. **if** $Rel(w) = null$ or ($Rel(w) = \leq$ and $update = <$) **then** $Rel(w) := update$; add w to *Open* (maintaining the list ordered by rank);
21. **if** $s = v$ **then** $RelChain(s) := \leq$ **else** $RelChain(s) := Rel(s)$;
22. $s := Pop(Open)$; add s to *Visited*;
23. $v := PrevOut(v)$; $RelChain(v) := null$;
24. For each node n in *Visited*, set $Rel(n)$ to *null*;

Fig. 4. Algorithm for computing the chain closure of a timegraph. We assume that $Rank(null) = +\infty$. $\hat{E}$ is the set of metaedges. $NG(s)$ abbreviates $Nextgreater(s)$. The *RelChain* and *Rel* flags, which we assume are initially set to *null*, are described in the text. Condition $s < w$ at step 19 can be checked in constant time by querying the timegraph because s and w belong to the same chain.

numbering, the *Rel* flag, and an additional information called *RelChain*. As mentioned above, when the search visits a node s with an outgoing metaedge on a chain already processed, i.e., $Chain(s) < Chain(v)$, there is no need to add the child nodes of s to *Open*. Similarly, when for a child node w we have $Rel(w) = <$, or $Rel(w) = \leq$ and $Rel(s) = \leq$, there is no need to add w to *Open*.

The meaning of the *RelChain* flag and its use for pruning the search at step 17 of the algorithm are slightly less intuitive. $RelChain(s) \in \{<, \leq, null\}$ indicates whether the chain of s has already been reached by the current search from v and, if $RelChain(s) \neq null$, it gives the strongest relation between v and *any* visited metanode on $Chain(s)$ that has been computed. When at step 17 the algorithm considers the s' nodes that are successors of s on the same chain as s, such nodes are added to *Open* only if $RelChain(s)$ is *null*, or $RelChain(s)$ is $\leq$ and $Rel(s)$ is $<$. In the other cases the s' nodes have already been considered by the search from v, and the current path from v to s' is not stronger than the one already found.

Finally, the elements in *Open* are maintained ordered by increasing rank, and the next visited node (step 22) is always one with the lowest rank in *Open*. This determines that, when a node s is visited, s cannot precedes any other node on the same chain as s that has already been visited by the current search; this property guarantees the correctness of the pruning described above.

The following theorem states the time complexity of ChainClosure.

Theorem 1. *Let T be a timegraph with $\hat{n}$ metanodes, $\hat{e}$ metaedges and n_c chains. The time complexity of Algorithm* ChainClosure *applied to T is $O(n_c \cdot \hat{e} + \hat{n}^2)$.*

Proof. The time complexity is determined by (1) the total number of metanodes and metaedges visited by all the searches from the metanodes with an outgoing cross-chain edge; (2) the total cost of maintaining *Open* ordered by the rank of its members.

Concerning (1), for each chain c, we have that the total number of metanodes entering into *Open* for *all* the searches from a metanode on c is $O(\hat{n})$. This is because, for each search from a metanode v on c, by exploiting the *Rel* flag (steps 19-20), a node s that has already been visited by the current or any previous search (from another metanode on c successor of v) enters into *Open* only if it is revisited with a stronger relation (step 20). It follows that, for each chain, the total number of visited metanodes is $O(\hat{n})$ and the total number of visited cross-chain edges is $O(\hat{e})$. Hence the total number of nodes and cross-chain edges visited by the algorithm is $O(n_c \cdot \hat{n} + n_c \cdot \hat{e}) = O(n_c \cdot \hat{e})$.

The other metaedges visited by the search are the *NextOut* links connecting two metanodes on the same chain. The total number of these edges in a timegraph is $O(\hat{n})$. By exploiting the *Rel* and *RelChain* flags (step 17), for each search, the algorithm visits each *NextOut* link at most twice. Hence, since the total number of searches performed by the algorithm is $O(\hat{n})$, the total number of visited *NextOut* links is $O(\hat{n}^2)$.

Concerning (2), consider the collection of the $O(\hat{n})$ nodes entering into *Open* for all searches from a start vertex on the same chain. By representing *Open* as a vector of metanodes indexed by their rank divided by the rank increment, and the fact that the nodes in *Open* are processed by increasing rank order, we can derive an efficient method for maintaining *Open* ordered in $O(\hat{n})$ total time. Hence, the total cost for maintaining *Open* ordered is $O(n_c \cdot \hat{n})$. It follows that the total time complexity of the algorithm is $O(n_c \cdot \hat{e} + \hat{n}^2)$. □

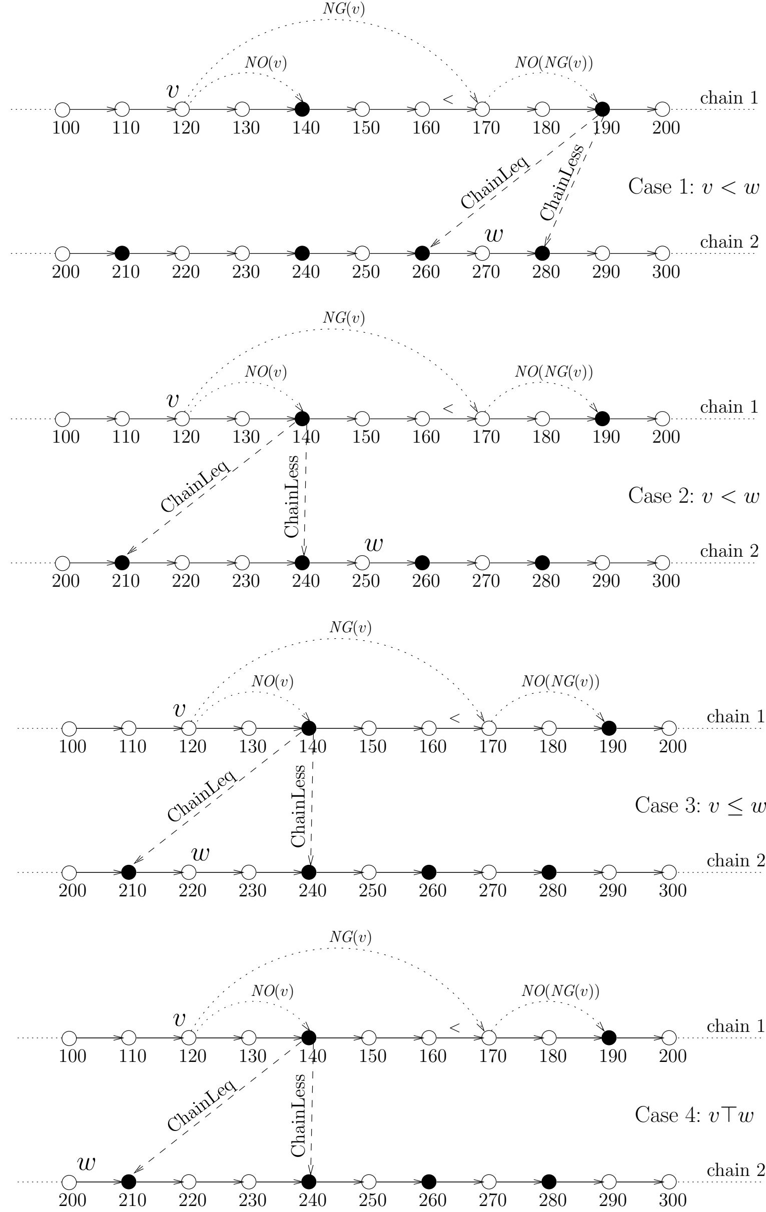

Fig. 5. Illustration of cases (1)–(4) in the constant time query algorithm outlined in the proof of Theorem 2. *NG* and *NO* abbreviate *Nextgreater* and *NextOut*, respectively. Edges on the same chain without a label are assumed to be $\leq$-edges.

3.2 Constant Time Queries and Minimal Constraints

In order to obtain a constant time query algorithm for every pair of variables, we exploit the *ChainLess* and *ChainLeq* information. Moreover, we assume that every node of the timegraph has a (possibly null) *NextOut* link associated with it.[4]

[4] This can easily be computed by post-processing the timegraph in linear time with respect to the number of nodes and edges in the timegraph. The *NextOut* links for

Theorem 2. *Let v and w be two point variables represented in a timegraph T extended with the chain closure. The strongest entailed relation between v and w can be computed in constant time.*

Proof (sketch). If v and w are on the same time chain, T entails $v = w$ or T entails $v \neq w$, then the strongest relation between v and w can be computed in constant time as described in [3].

If v and w are on different chains, the strongest relation can be computed by exploiting the chain closure as follows. Without loss of generality, suppose that v is on chain 1 and w is on chain 2. There are three cases to consider: (a) none of v and w are metanodes, (b) one of v and w is a metanode, (c) both v and w are metanodes. In the rest of the proof we only consider case (a). Cases (b) and (c) can be handled in a similar, slightly simpler, way.

If $NextOut(v) = null$, then clearly the query answer is "$\top$". Assume that $NextOut(v) \neq null$, and let $no = NextOut(v)$ and $ng = Nextgreater(v)$, if $Nextgreater(v)$ has a cross-chain outgoing edge, $ng = NextOut(Nextgreater(v))$ otherwise. There are the following four cases to consider, which are illustrated in Figure 5:

(1) if $Rank(w) \geq Rank(ChainLeq(ng, 2))$, then the strongest entailed relation between v and w is "$<$";
(2) if case (1) does not apply and $Rank(w) \geq Rank(ChainLess(no, 2))$, then the strongest entailed relation is "$<$";
(3) if cases (1-2) do not apply and $Rank(w) \geq Rank(ChainLeq(no, 2))$, then the strongest entailed relation is "$\leq$";
(4) if cases (1–3) do not apply, the strongest entailed relation is "$\top$". □

Since, for any pair of variables in the timegraph T with chain closure representing a CSP C, the strongest entailed relation can be obtained in constant time, T can be considered as a representation of the minimal CSP of C. However, if we want an explicit representation of the minimal CSP, we need an additional step to read all minimal constraints from T, which can be done in quadratic time w.r.t. the number of variables in C.

The next theorems state the complexity of our method for computing the explicit minimal CSP for the convex PA and the full PA.[5]

Theorem 3. *Let C be a CSP over the convex Point Algebra involving n variables, the explicit minimal CSP of C can be computed in $O(\hat{n} \cdot \hat{e} + n^2)$ time, where $\hat{n}$ and $\hat{e}$ are the metanodes and metaedges, respectively, in the timegraph representing C.*

---

a node that is not a metanode are used only to support constant time queries as described in the proof of Theorem 2; they are not part of the metagraph. In the original version of the timegraph only the metanodes can have a *NextOut* link.

[5] If we use the extended timegraph as the representation of the output minimal constraints, then the $O(n^2)$ term in the complexity bounds can be omitted, and we need to add the number of input constraints as an additional term; such a number is no greater than $O(n^2)$ and for sparse CSPs is less than quadratic.

Proof. The total time complexity for constructing the timegraph for C is $O(n + |C| + \hat{n} \cdot \hat{e})$ [3], where $|C|$ is the number of constraints in C. By Theorem 2, computing the strongest relation entailed by a timegraph with chain closure for a pair of variables can be accomplished in constant time, and so all minimal constrains of C can be obtained in $O(n^2)$ time. By Theorem 1, the chain closure of the timegraph for C can be computed in $O(n_c \cdot \hat{e} + \hat{n}^2)$. It follows that the total time complexity of computing the minimal CSP of C is $O(\hat{n} \cdot \hat{e} + n^2)$. □

Theorem 4. *Let C be a CSP over the full Point Algebra involving n point variables, the explicit minimal CSP of C can be computed in $O(\hat{e} \cdot \hat{n} + \hat{e}_{\neq} \cdot (\hat{e} + \hat{n}) + n^2)$ time, where $\hat{n}$, $\hat{e}$ and $\hat{e}_{\neq}$ are the metanodes, metaedges and $\neq$-metaedges, respectively, in the timegraph representing C.*

Proof (sketch). The proof is similar to the proof of Theorem 3, except that the time complexity of computing the timegraph for C is $O(n+|C|+\hat{e}\cdot\hat{n}+\hat{e}_{\neq}\cdot(\hat{e}+\hat{n}))$. □

Remark. We observe that the number of metanodes and metaedges in a timegraph can be significantly smaller than the number of variables n and constraints c in the input CSP. For this reason, in practice the time complexity of our approach can be lower than the complexity of the known techniques for computing the minimal CSP of a CSP over the convex PA ($O(n^3)$) and the full PA ($O(n^3 + c_{\neq} \cdot n^2)$), where $c_{\neq}$ is the number of input $\neq$-constraints [10, 6]. Moreover, for a sparse input CSP, our method has a better worst-case time complexity. In particular, when c is linear with respect to n, the complexity of our method is $O(n^2)$, while the techniques in [10, 6] require $O(n^3)$ time. This can be shown by considering a CSP with n variables forming a single time chain. The path-consistency algorithm in [10] revises the relation between $O(n^2)$ pairs of variables, and for each of these pairs it considers $O(n)$ triples of variables.

4 Experimental Analysis

We implemented (in C) all timegraph algorithms described in [3] and our new chain closure algorithm. We call the resulting temporal reasoner TGC. In order to evaluate the effectiveness of our approach in practice, we compared the performance of TGC with an implementation of

- the path-consistency algorithm given in [10] with the improvements proposed in [8], which we call PC,
- PC extended with van Beek's algorithm for removing the forbidden graphs in a path-consistent CSP over PA [6], which we call PC-FG.[6]

[6] We carefully checked the code of PC and PC-FG to make the implementation efficient and bug-free. The correctness of the three implemented systems was checked by comparing the corresponding minimal constraints obtained for the same CSP, using many randomly generated CSPs.

PC solves the problem of computing the minimal CSP for the convex PA [9], while PC-FG solves this problem for the full PA [4, 6].

The output of PC and PC-FG is a matrix M such that each entry $M[i, j]$ contains a representation of the strongest entailed relation (minimal constraint) between the i-th and the j-th variables in the input CSP. The output of TGC is the collection of data structures representing a timegraph with chain closure. In both cases, the minimal constraint for a pair of variables can be read from the corresponding data structures in constant time. In our experimental comparison, for TGC we also consider the additional total CPU-time for making all possible queries, which for PC and PC-FG was not considered.

4.1 Experimental Settings and Test Domains

The experiments were aimed at evaluating the hypothesis that our approach works well in practice for chain-structured CSPs (common, e.g., in automated planning and story comprehension) as well as for randomly generated (sparse and dense) CSPs. The data sets were obtained by running three different generators, two of which are based on the available code of known generators [1, 3]. Since we were mostly interested in testing the use of our closure algorithm, every generated CSP is consistent and does not contain or entail equality constraints.

In each plot with the experimental results, for the CPU-time of TGC we consider two series of experimental results: one includes the total time for performing all queries, the other does not. In both cases the CPU-time for TGC includes the construction of the timegraph and the chain closure. All tests were conducted on an Intel Xeon(tm) 3 GHz, 1 Gbytes of RAM. The CPU-time corresponding to each point on a curve is an average value over 100 CSPs.

We used six test domains. The CSPs in Domains A, B, C and E, were synthesized using DelGrande et al.'s generator [1], while Domain D using Gerevini & Schubert's generator [3]. For Domain F we used a new generator.

Domain A: randomly generated CSPs over the convex PA with equal numbers of $<$, $\leq$-constraints and no $\neq$-relation. For each considered number n of variables, 100 CSPs with $n \cdot \lfloor(\log_2(n))\rfloor$ constraints each are generated.

Domain B: this domain is the same as Domain A, except that the differences among the CSP instances concern the number c of constraints, rather than the number of variables, which is set to 1000. The generated CSPs range from sparse CSPs ($c = 3000$) to dense CSPs ($c = 449, 850$).

Domain C: randomly generated CSPs over the convex PA with a chain based structure. For each considered number n of variables, the procedure generates 100 CSPs as described in [1].

Domain D: randomly generated CSPs over the convex PA consisting of data sets that tend to fall into chains. For each considered number n of variables, the procedure generates 100 CSPs as described in [3], each of which contains $n \cdot \lfloor(\log_2(n))\rfloor$ constraints.

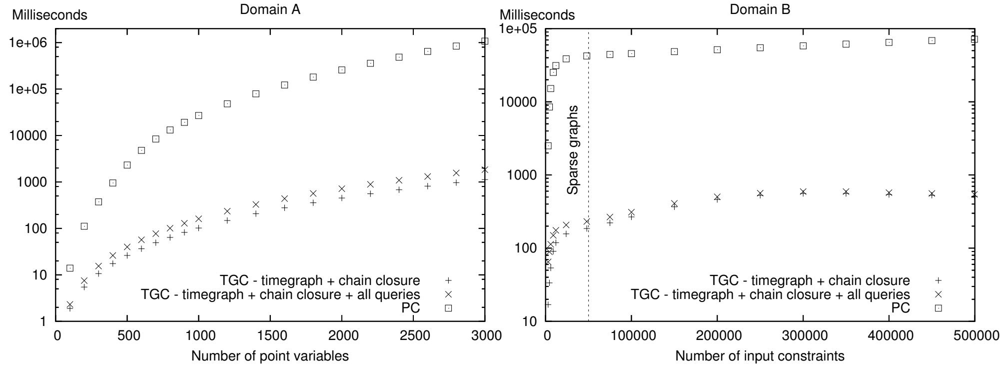

Fig. 6. Average CPU-times of TGC and PC in Domains A and B. For Domain A, on the x-axis we have the number of point variables, while for Domain B we have the number of edges (input constraints) in the timegraph. In both cases on the y-axis we have the CPU-milliseconds (logarithmic scale).

Domain E: this domain is the same as Domain A, except that the CSPs also contain $\neq$-constraints. For each considered number n of variables, the procedure generates 100 CSPs. Each generated CSP contains $n \cdot \lfloor(\log_2(n))\rfloor$ constraints, 10% of which are $\neq$-constraints.

Domain F: randomly generated CSPs over the full PA consisting of CSPs with a chain structure and many $\neq$-diamonds (forbidden graphs). For each considered number n of variables, the procedure generates 100 CSPs as follows. We partition the variables in three subsets, each of which containing $\lfloor n/3 \rfloor$ variables constrained to form a chain of $\leq$-constraints. The variables in each subset are constrained in a way that determine at least $\lfloor n/3 \rfloor - 1$ $\neq$-diamonds. In each generated CSP, the percentage of $\neq$-constraints is about 50.

4.2 Experimental Results

First we present the results for the convex PA and then those for the full PA. Figure 6 shows the performance of TGC and PC for Domains A and B. The results for Domain A (plot on the left side of the figure) indicate that for large CSPs TGC is up to about three orders of magnitude faster than PC. The results for Domain B (plot on the right side of the figure) indicate that the very good performance of TGC relative to PC does not depend on the sparseness of the input CSP: regardless the sparseness of the CSP, TGC is always much faster than PC.

Figure 7 shows the performance of the compared approaches in Domains C and D (sparse CSPs which exhibit chain structure). In these domains the performance gap is even more dramatic than for randomly generated CSPs, obtaining an improvement of up to four orders of magnitude, without considering the CPU-time for performing all queries, and three orders, considering all queries.

Figure 8 concerns CSPs over the full PA, i.e., Domains E and F. TGC is up to two orders of magnitude faster than PC-FG. Here the improvements are

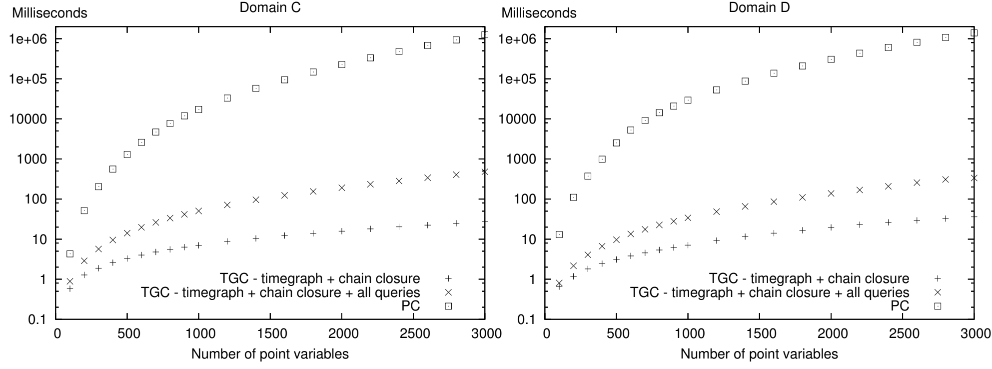

Fig. 7. Average CPU-times of TGC and PC in Domains C and D. On the x-axis we have the number of point variables, on the y-axis the CPU-milliseconds (logarithmic scale).

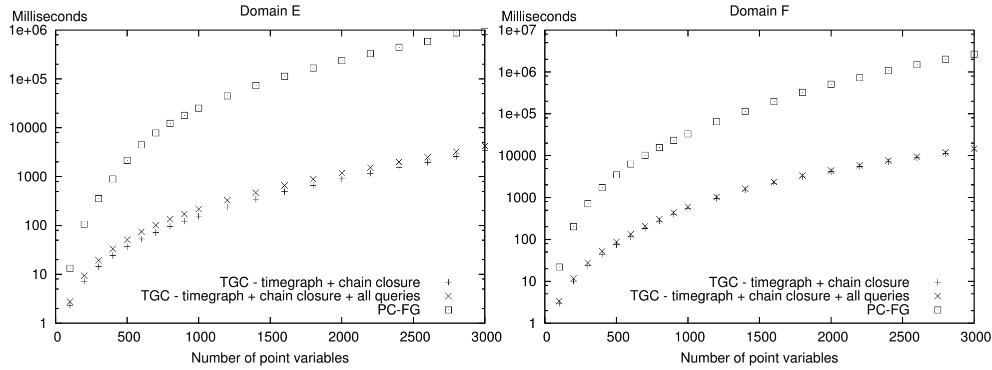

Fig. 8. Average CPU-times of TGC and PC-FG in Domains D and E. On the x-axis we have the number of point variables, on the y-axis we have the CPU-milliseconds (logarithmic scale).

less dramatic (but still very significant), because of the cost of dealing with $\neq$-constraints, which can be the most expensive processing step both in the construction of the timegraph with chain closure and in PC-FG.

Finally, we also tested a simple alternative method for computing the minimal CSP using the timegraph representation. Instead of computing the chain closure, we use the original data structures and query algorithms to compute, for every pair of variables, the corresponding strongest entailed relation. We compared the CPU-times of this simple approach and the one computing the chain closure using our test domains. The results of this experiment indicate that computing the chain closure is advantageous. Our method is up to 300 times faster than making all queries in the timegraph without chain closure. An exception to this behavior are particular CSPs that have a very strong chain structure (e.g., when the timegraph has only one chain), where an extremely high percentage of pair

of nodes lie on the same chain. In these special cases the two timegraph-based approaches perform similarly.

5 Conclusions

We have presented a new efficient method for computing the minimal CSP of a CSP over the Point Algebra. For sparse CSPs the worst-case complexity of our method improves the complexity of known methods based on enforcing path consistency. In practice, an experimental analysis using various types of data sets shows that our approach is much faster than computing the minimal CSP using known techniques, both for CSPs with a chain structure and for general randomly generated (sparse and dense) CSPs. A price to pay for the efficiency gain is the more complex algorithms and implementation (however, the implementation used in our experiments will be made publicly available).

While the techniques presented in this paper build on the timegraph representation, we believe that our method can be applied to other graph-based representation, such as the metagraph of the SPMG system [1], which uses series-parallel graphs instead of chains. Future work includes investigating an algorithm for the metagraph closure in the context of SPMG and additional experiments.

Acknowledgments. We would like to thank Nicola Ferrari and Nicola Morotti for their help with the implementation of the timegraph algorithms.

References

1. J. Delgrande, A. Gupta, and T. Van Allen. A comparison of point-based approaches to qualitative temporal reasoning. *Artificial Intelligence*, 131:135–170, 2001.
2. A. Gerevini. Processing qualitative temporal constraints. In *Handbook of Temporal Reasoning in Artificial Intelligence*, pages 247–276. Elsevier, 2005.
3. A. Gerevini and L. Schubert. Efficient algorithms for qualitative reasoning about time. *Artificial Intelligence*, 74:207–248, 1995.
4. A. Gerevini and L. Schubert. On computing the minimal labels in time point algebra networks. *Computational Intelligence*, 11(3):443–448, 1995.
5. C.M. Golumbic and R. Shamir. Complexity and algorithms for reasoning about time: a graph-theoretic approach. *Journal of the Association for Computing Machinery (ACM)*, 40(5):1108–1133, 1993.
6. P. van Beek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58(1-3):297–321, 1992.
7. P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–144, 1990.
8. P. van Beek and D.W. Manchak. The design and experimental analysis of algorithms for temporal reasoning. *Journal of Artificial Intelligence Research*, 4:1–18, 1996.
9. M. Vilain and H.A. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the Fifth National Conference of the American Association for Artificial Intelligence (AAAI-86)*, pages 377–382. Morgan Kaufmann, 1986.
10. M. Vilain, H.A. Kautz, and P. van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. In D.S Weld and J. de Kleer, editors, *Readings in Qualitative Reasoning about Physical Systems*, pages 373–381. Morgan Kaufmann, San Mateo, CA, 1990.