

Bi-Intervals for Backtracking on Temporal Constraint Networks

Jean-François Baget
INRIA Rhône-Alpes and LIG
Montbonnot, France
jean-francois.baget@inrialpes.fr

Sébastien Laborie
INRIA Rhône-Alpes and LIG
Montbonnot, France
sebastien.laborie@inrialpes.fr

Abstract

Checking satisfiability of temporal constraint networks involves infinite variables domains. We explore a solution based upon finite partitions of infinite domains. Though a straightforward partition results in a sound and complete backtrack, its extension to forward checking is not complete. Using bi-intervals, we obtain sound and complete backtrack and forward checking algorithms. Moreover, we show that bi-intervals used in a hybrid algorithm which also instantiates constraints improve backtrack efficiency.

1. Introduction

To check satisfiability of a temporal constraint network (TCN) [5] (variables domains are the sets of intervals of real numbers and constraints are disjunctions of Allen's relation [1]), most algorithms instantiate constraints and use path-consistency based techniques to evaluate solutions [10].

Some authors suggested (e.g., [13]) to use generic constraint optimization techniques (e.g., forward checking [8], backjumping [6, 4, 11]). However, these optimizations instantiate variables which have an infinite domain in TCNs. Backtracking on the dual graph [12] is a solution but it does not handle mixed qualitative/quantitative TCNs [13]. For mixed TCNs, one can rely upon finite domains of equivalence classes of intervals (that we name abstract intervals), a natural encoding similar to the interval end-point ordering of [15]. We provide in Section 3 a sound and complete TCN satisfiability algorithm using abstract intervals. Though it remains sound and complete using backjumping schemes, we show that its extension to a simple look-ahead technique such as forward checking is not complete.

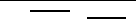
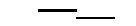
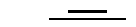
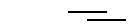
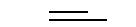
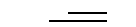
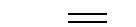
To obtain a sound and complete forward checking, we propose in Section 4, a different equivalence class called bi-intervals (intervals of intervals). Using them, both backtrack and forward checking are sound and complete. Finally, Section 5 compares the efficiency of our algorithms with those based upon path-consistency [10].

2. Allen Temporal Constraint Networks

In this section, we define temporal constraint networks: constraints are disjunctions of Allen's relations [1] and variable values can be any interval of real numbers. We formally define these objects and discuss existing algorithms that check their satisfiability.

2.1. Allen's relation

The Allen interval algebra [1] is based on the notion of time intervals and binary relations on them. A time interval I is an ordered pair (I_s, I_e) of real numbers such that $I_s < I_e$. Allen defined a set \mathcal{R} of basic relations between intervals such that any pair of time intervals satisfies one and only one of these relations:

relation (r): $x r y$	x / y	inverse: $y r^{-1} x$
before (b)		(bi) after
meets (m)		(mi) met-by
during (d)		(di) contains
overlaps (o)		(oi) overlapped-by
starts (s)		(si) started-by
finishes (f)		(fi) finished-by
equals (e)		(e)

2.2. Temporal Constraint Networks

Definition 1 (Temporal constraint network (TCN)) Let \mathcal{D} be a domain containing all time intervals, and \mathcal{R} a set of Allen relations over \mathcal{D} . A temporal constraint network (TCN) over $(\mathcal{D}, \mathcal{R})$ is a tuple $\mathcal{N} = (V, U, \delta, \rho)$ where V is a set of variables, $U \subseteq V \times V$ is a set of constraints, $\delta : V \rightarrow \mathcal{P}(\mathcal{D})$ maps each variable to a set of values and $\rho : U \rightarrow \mathcal{P}(\mathcal{R})$ maps each constraint to a set of relations.

Definition 2 (Solution of a TCN) Let $\mathcal{N} = (V, U, \delta, \rho)$ be a TCN over $(\mathcal{D}, \mathcal{R})$. A solution of \mathcal{N} is a mapping $\alpha : V \rightarrow \mathcal{D}$ such that $\forall v \in V, \alpha(v) \in \delta(v)$ and $\forall \langle v_1, v_2 \rangle \in U, \exists R \in \rho(\langle v_1, v_2 \rangle)$ with $\langle \alpha(v_1), \alpha(v_2) \rangle \in R$.

A TCN is said *satisfiable* if it admits a *solution*.

Example 1 The TCN of Fig. 1 is *satisfiable*: it admits the solution $I = [\pi/2, 3]$, $J = [3, \sqrt{29}]$ and $K = [1, 4]$.

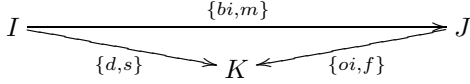


Figure 1. An example of TCN.

2.3. Algorithms for TCNs Satisfiability

A TCN is *basic* if each of its constraints contains only one Allen relation. To a TCN \mathcal{N} we can associate an exponential set $B(\mathcal{N})$ of $\prod_{i=1}^{|U|} |\rho(u_i)|$ of basic TCNs such that \mathcal{N} is satisfiable if and only if one of the networks in $B(\mathcal{N})$ is path-consistent [1]. This algorithm generates all instantiations of constraints and tests them using path-consistency (sound, but complete only for basic networks). As an optimization heuristic, [10] proposed path-consistency as a filtering technique during backtrack, pruning significantly the search tree. Additional optimizations include constraints and relations orderings as well as optimizations of path-consistency (e.g., [10, 16, 13]).

However, these algorithms do not instantiate variables but constraints. To adapt them to a variable-instantiating framework [13] and to process mixed qualitative/quantitative temporal networks, [9] represents TCNs as constraint networks whose variables are either time intervals or points representing the end points of these intervals. He then uses specific path-consistency algorithms on these networks. Though many authors (e.g., [13]) point out that it is interesting to optimize the backtracking part of these algorithms with generic backtrack optimizations (such as look-ahead or backjumping schemes [6, 4, 11]), we are not aware of any implementation of such a hybrid algorithm.

Though we are mainly interested here in sound and complete algorithms, we must include in this quick overview of algorithms the efficient, but not complete, local search algorithm of [15] that also instantiates variables of the network.

3. I-BT: Refining Intervals during Backtrack

We want to use a backtracking algorithm to check the satisfiability of a TCN. However, domains of variables (pairs of real numbers) are infinite leading to an infinite backtrack. We first define abstract intervals that will be used to finitely partition the solutions of the network. They are a natural encoding of interval end-points also used in [15]. Then, we present a backtrack algorithm **BT** and show how to adapt it to run on these abstract intervals. Finally, we show that an extension of **BT** to forward checking [8] fails on abstract intervals.

3.1. Abstract Intervals

Definition 3 (Abstract interval) Let L be a totally ordered list. An abstract interval over L is a pair noted $[a, b]$ of elements of L s.t. $a < b$.

By instantiating variables with abstract intervals, we compute equivalence classes of solutions (in the sense of Def. 2): two solutions α, α' are in the same equivalence class iff for every pair of variables $\langle v_1, v_2 \rangle$, the Allen relation that holds between $\alpha(v_1)$ and $\alpha(v_2)$ is the same relation that holds between $\alpha'(v_1)$ and $\alpha'(v_2)$.

Definition 4 (Abstract solution) Let $\mathcal{N} = (V, U, \delta, \rho)$ be a TCN. An abstract solution of \mathcal{N} is a pair (L, α) where L is an ordered set of elements (a list), and α is a surjective mapping from V to L^2 such that the mapping α' defined by $\forall x \in V$ with $\alpha(x) = \langle s, e \rangle, \alpha'(x) = [\text{rank}_L(s), \text{rank}_L(e)]$ is a solution of \mathcal{N} (where $\text{rank}_L(p)$ returns the position of an element p in the list L).

Proposition 1 Abstract solutions of a TCN are equivalence classes for its solutions.

3.2. Backtracking with abstract intervals

Backtrack [7] can be used to check the satisfiability of a TCN. As in [11], the iterative version proposed in Alg. 1, denoted **BT**, allows a better control of the execution stack.

Algorithm 1: Backtrack

Data: A constraint network $\mathcal{N} = (V, U, \delta, \rho)$ over $(\mathcal{D}, \mathcal{R})$ (\mathcal{D} finite and $r \in \mathcal{R}$ decidable).
Result: YES iff \mathcal{N} is satisfiable.
if $V = \emptyset$ **then return** SolutionFound(\mathcal{N});
 $\mathcal{N} \leftarrow$ InitializeNetwork(ReorderVariables(\mathcal{N}));
backtrack \leftarrow false; level \leftarrow 1;
while level \neq 0 **do**
 if level = $|V| + 1$ **then**
 SolutionFound(\mathcal{N});
 backtrack \leftarrow true;
 level \leftarrow PreviousLevel(level, \mathcal{N});
 else currentVariable \leftarrow V[level];
 if backtrack **then**
 if NextCandidate(currentVariable, \mathcal{N}) **then**
 backtrack \leftarrow false;
 level \leftarrow NextLevel(level, \mathcal{N});
 else level \leftarrow PreviousLevel(level, \mathcal{N});
 else
 if FirstCandidate(currentVariable, \mathcal{N}) **then**
 level \leftarrow NextLevel(level, \mathcal{N});
 else
 backtrack \leftarrow true;
 level \leftarrow PreviousLevel(level, \mathcal{N});

BT accepts any variable ordering of *ReorderVariables*, though some are more efficient (e.g., graph traversal).

InitializeNetwork stores data for quicker later accesses (e.g., *pre* and *post* lists). The integer *level* is the current depth in the search tree and the boolean *backtrack* is FALSE when going down the search tree and TRUE otherwise. Here, *NextLevel* and *PreviousLevel* return respectively $level + 1$ and $level - 1$ ¹. *SolutionFound* is called when all variables are instantiated. For satisfiability, it returns YES, and stops **BT**. The heart of **BT** is encoded in *FirstCandidate* and *NextCandidate*:

Candidates Consider the variables v_1, \dots, v_k instantiated with $\alpha(v_1), \dots, \alpha(v_k)$. *Candidates* of v_{k+1} are all values $a \in \delta(v_{k+1})$ such that for every constraint $c = \langle v_i, v_{k+1} \rangle$ in \mathcal{N} with $1 \leq i \leq k$ (resp., $c = \langle v_{k+1}, v_i \rangle$), $\langle \alpha(v_i), a \rangle \in \rho(c)$ (resp., $\langle a, \alpha(v_i) \rangle \in \rho(c)$): we say that a is *compatible* with v_i . For **BT** to be sound and complete, *FirstCandidate* must compute the first candidate of the current variable, store it in *currentCandidate*, return TRUE if found or FALSE otherwise. Successive calls to *NextCandidate* iterate through candidates, updating *currentCandidate*, return TRUE if found or FALSE otherwise.

Pre and post variables We call $pre_V(v)$ the list of neighbors of the variable v that are smaller than v according to the order of *ReorderVariables*. Only constraints incident to v and a variable of $pre_V(v)$ are needed to compute candidates for v . The list $post_V(v)$ is defined using neighbors greater than v .

Refinement lists Each variable v contains a sequence of *refinement lists*. If $pre_V(v) = \emptyset$, this sequence consists of a single list $\lambda(v) = \delta(v)$. Otherwise, it contains $|pre_V(v)|$ lists $\lambda(v, v_1), \dots, \lambda(v, v_p)$ where $pre_V(v) = (v_1, \dots, v_p)$. **BT** computes these lists for the current variable v each time *FirstCandidate* is called. The list $\lambda(v, v_1)$ contains all values of $\delta(v)$ compatible with v_1 . For $2 \leq i \leq p$, the list $\lambda(v, v_i)$ contains all values of $\lambda(v, v_{i-1})$ compatible with v_i . Then the list $\lambda(v, v_p)$ contains all candidates of v .

During **BT**, we maintain a list *pointsList* that respects the following property: “*pointsList* contains only the start and end points of instantiated variables”. We call **I-BT** (Interval BackTrack) the specialization of **BT** (Alg. 1) that calls Alg. 2 to check satisfiability. It only generates the abstract intervals that are required during **I-BT**. While building refinement lists, we use an encoding of abstract intervals (refinement intervals) that implicitly represents them without updating *pointsList*.

Refinement intervals A refinement interval over *pointsList* is a pair $\langle S, E \rangle$ where $S = +a$ or $S = a$ and $E = -b$ or $E = b$, with $[a, b]$ an abstract interval over *pointsList*. Refinement intervals are used to build the refinement lists.

¹Forward jumping [2] or backjumping [6, 4, 11] allow greater increments or decrements of *level*.

Algorithm 2: FirstCandidate (I-BT version)

Data: A TCN $\mathcal{N} = (V, U, \delta, \rho)$ over $(\mathcal{D}, \mathcal{R})$, where the variables v_1, \dots, v_k have been instantiated with pairs of points of *pointsList*; and the variable v_{k+1} .

Result: Computes the candidates of v_{k+1} , returns FALSE if this list is empty, otherwise returns TRUE and *currentCandidate* points on the first element of this list.

```

if  $pre_V(v_{k+1}) = \emptyset$  then
   $\lambda(v_{k+1}) \leftarrow \text{GenerateAllIntervals}(\text{pointsList});$ 
   $\text{currentCandidate} \leftarrow \text{FirstElement}(\lambda(v_{k+1}));$ 
   $\text{Instantiate}(\text{currentCandidate});$ 
  return TRUE;
else
   $(v'_1, \dots, v'_p) \leftarrow pre_V(v_{k+1});$ 
  for  $i \in \{1, \dots, p\}$  do
     $\lambda(v_{k+1}, v'_i) \leftarrow \text{EmptyList}();$ 
   $\lambda(v_{k+1}, v'_1) \leftarrow \text{GenerateAllIntervals}(\text{pointsList}, v'_1);$ 
  for  $i \in \{2, \dots, p\}$  do
    for  $a \in \lambda(v_{k+1}, v'_{i-1})$  do
      if  $IsCompatible?(a, v'_i)$  then
         $\lambda(v_{k+1}, v'_i) \leftarrow \text{AddToList}(\lambda(v_{k+1}, v'_i), a);$ 
  if  $\lambda(v_{k+1}, v_p) = \emptyset$  then
    return FALSE;
  else
     $\text{currentCandidate} \leftarrow \text{FirstElement}(\lambda(v_{k+1}, v_p));$ 
     $\text{Instantiate}(\text{currentCandidate});$ 
    return TRUE;

```

When choosing a refinement interval $\langle S, E \rangle$ as candidate in the last refinement list, *Instantiate* generates the equivalent abstract interval $[a, b]$, where $a = x$ if S is an element x of *pointsList* or a is a new element immediately following x in *pointsList* if $S = +x$, and $b = y$ if E is an element y of *pointsList* or b is a new element immediately preceding y in *pointsList* if $S = -y$. Note that *Instantiate* adds 0, 1 or 2 elements to *pointsList* (these elements will be removed during each backtrack).

At the beginning of **I-BT**, *pointsList* is initialized with two elements (A, Z) where A and Z respectively stand for $-\infty$ and $+\infty$. *GenerateAllIntervals* builds the first refinement list $LR(v)$ of each variable v . If $pre_V(v) = \emptyset$, $LR(v)$ contains for each abstract interval $[a, b]$ over the current *pointsList* the refinement intervals $\langle a, b \rangle$, $\langle +a, b \rangle$, $\langle a, -b \rangle$ and $\langle +a, -b \rangle$. Otherwise, $LR(v)$ contains the abstract intervals of the previous construction that are compatible with the current candidate of the first variable in $pre_V(v)$. Note that, to avoid the unconstrained generation of refinement intervals in $LR(v)$ (a quadratic number in the size of *pointsList*), we implemented 13 different specialized functions, one for each Allen relation, that directly generate the required refinement intervals in $LR(v)$. It is immediate to check that **I-BT** enumerates exactly the same candidates as the naive **BT** with quadratic domain.

Proposition 2 **I-BT** is sound and complete w.r.t. the set of abstract solutions.

Fig. 2 shows the search tree of **I-BT** on Ex. 1.

ReorderVariables provides the variable ordering (I, J, K) . Each node represents refinement lists, that contain refinement intervals. Each edge is labeled by the current state of *pointsList* and the current candidates (the abstract intervals) of the instantiated variables are represented at the bottom of these lists. On this example, **I-BT** returns TRUE, i.e., the TCN is satisfiable. However, we also present in Fig. 2 all other solutions of the TCN for a better understanding of refinement intervals (only a slight adjustment of Alg. 1 is required to enumerate or count all solutions of a TCN).

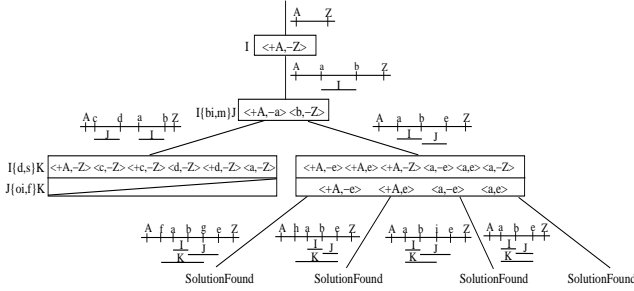


Figure 2. Backtrack tree of I-BT.

3.3. Problems with forward checking

Forward Checking (FC) [8] is an enhancement easily implemented with refinement lists. In **I-BT**, refinement lists of v are computed when v is explored. However, the refinement list $\lambda(v, v_i)$ can be computed as soon as v_i is explored. In that case, refinement lists of v are already built when exploring v : we only have to iterate through the candidates and propagate their choice to the variables in $post_V(v)$. If a refinement list in some $v'_i \in post_V(v)$ is emptied by propagation, then we choose another candidate. This method avoids backtracks across all variables between v and v'_i .

Though **I-BT** is more efficient than the naïve **BT** with quadratic domains, the data structure we use does not allow extension to FC, although this extension is in general sound and complete. The reason is that **I-BT** computes candidates according to a list of instantiations *pointsList*. Hence, an extension to FC should compute candidates according to this list. However, let us explore a variable v having two predecessors in $pre_V(v)$, v_1 and v_2 . The two refinement lists of v are function of the current state of *pointsList*, itself dependent upon the instantiations of both v_1 and v_2 . Suppose we backtrack and change the instantiation of v_2 . FC relies on the fact that the refinement intervals in the first refinement list (built from v_1) are still valid; but this is false in this case, since this list depends upon the instantiation of v_2 . This problem is easy to identify on Fig. 2. The first refinement lists of the third level corresponding to $I\{d, s\}K$ contain different references to points in *pointsList* depending on the instantiation of the refinement interval corresponding to J .

4. BI-BT: Refining Bi-Intervals

To upgrade our algorithm to forward checking, we have to ensure that the refinement list of a variable v built from the instantiation of one of its predecessors v' depends only on the elements of *pointsList* present during the exploration of v' . This is the goal of the following version that relies on bi-intervals (intervals of intervals) in the refinement lists. Bi-intervals have the same function as refinement intervals in **I-BT**. However, they encode sets of abstract intervals (a bi-interval can be instantiated by many abstract intervals).

4.1. Bi-intervals

A *bi-interval* over a list L is a pair $\langle \dagger_1 a, b \dagger_2, \dagger_3 c, d \dagger_4 \rangle$ where the symbols \dagger_i stand for the usual interval delimiters [or], a, b, c and d are elements of L , and the maximal element of $\dagger_3 c, d \dagger_4$ is greater than the minimal element of $\dagger_1 a, b \dagger_2$. Bi-intervals encode sets of abstract intervals.

Definition 5 (Interpretation of a bi-interval) Let $B = \langle \dagger_1 a, b \dagger_2, \dagger_3 c, d \dagger_4 \rangle$ be a bi-interval over L . The interpretation of B in L is the set of all abstract intervals $[x, y]$ such that $x \in \dagger_1 a, b \dagger_2$ and $y \in \dagger_3 c, d \dagger_4$ and either:

- both x and y are elements of L ;
- x or y is an element of L and the other is a new element between two contiguous elements of L ;
- x and y are two new elements placed between two contiguous elements of L ;
- x and y are two new elements placed between two distinct pairs of contiguous elements of L .

The next definition, along with Prop. 3, states that it is possible to build successive refinement lists using bi-intervals: the union of the interpretations of bi-intervals in the last refinement lists corresponds to candidates in **I-BT**.

Definition 6 (Refining a bi-interval) Let I be an abstract interval, B a bi-interval over L and r an Allen relation. The refinement of B following I and r is the subset of the interpretation of B in L composed of all abstract intervals J such that $\langle I, J \rangle \in r$.

Proposition 3 Let I be an abstract interval, B a bi-interval over L and r an Allen relation. The refinement of B following I and r is either the empty set or the interpretation of a bi-interval B' over L .

Our implementation is a proof of that proposition. We implemented 13 different refining functions (one for each Allen relation) and enumerated in these functions all possible interactions between an abstract interval and a bi-interval. Prop. 3 is always verified. Alg. 3 is an example of these 13 functions for the *finished-by* relation.

Algorithm 3: Refining a bi-interval constrained by the *finished-by* relation

Data: A bi-interval $I = \langle \dagger_1 a, b \dagger_2, \dagger_3 c, d \dagger_4 \rangle$, and an abstract interval $J = [x, y]$.

Result: Let R be the refinement of I following J and the *finished-by* relation. We return FALSE if R is empty or a bi-interval whose interpretation in *pointsList* is R otherwise.

if $y \notin \dagger_3 c, d \dagger_4$ then return FALSE;
 if $b \leq x$ then return FALSE;
 if $a \geq y$ then return FALSE;
 if $(a \leq x \text{ and } b \geq y)$ then return $\langle x, y, [y, y] \rangle$;
 if $(a \leq x \text{ and } b < y)$ then return $\langle x, b \dagger_2, [y, y] \rangle$;
 if $(a > x \text{ and } b \geq y)$ then return $\langle \dagger_1 a, y, [y, y] \rangle$;
 if $(a > x \text{ and } b < y)$ then return $\langle \dagger_1 a, b \dagger_2, [y, y] \rangle$;

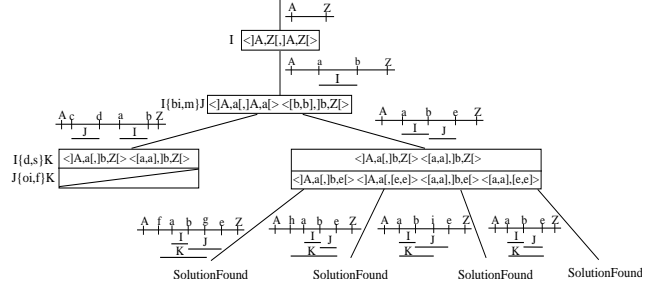


Figure 3. Backtrack tree of BI-BT.

4.2. Backtracking with bi-intervals

In **BT**, enumeration of candidates starts with a given finite domain. Each successive refinement list contains a smaller subset of this domain and finally the last refinement list contains the candidates. In **I-BT**, the procedure is similar but begins with a generation of the abstract intervals required in the first refinement list (*GenerateAllIntervals*). The size of the first refinement list can be quadratic in the size of *pointsList*.

Our version of **BT** that relies on bi-intervals is called **BI-BT**. This time, the domain of a variable v is represented by a single bi-interval $\langle]A, Z[,]A, Z[\rangle$ (as in **I-BT**, A and Z are initial elements of *pointsList* that stand for $-\infty$ and $+\infty$). If $pre_V(v) = \emptyset$, the refinement list $\lambda(v)$ contains this single bi-interval (instead of the quadratic number in **I-BT**). Though in **BT** and **I-BT** each refinement list contained a subset of the previous one, refinement lists *grow* in **BI-BT**. For each bi-interval $B \in \lambda(v, v_k)$ and for each Allen relation $r \in \rho(\langle v, v_{k+1} \rangle)$ (or $\rho(\langle v_{k+1}, v \rangle)$), $\lambda(v, v_{k+1})$ contains the bi-interval representing the refinement of B following the current candidate of v_{k+1} and r (see Def. 6 and Prop. 3), if this set is not empty. A refinement list of size k can thus be followed by a refinement list $\lambda(v, v')$ of size at most $k \times |\rho(\langle v, v' \rangle)|$. Finally, the last refinement list contains bi-intervals but the candidates we look for are abstract intervals: the set of candidates we must iterate through is then the union of the interpretations of these bi-intervals in *pointsList*.

Prop. 3 ensures that the candidates chosen in **BI-BT** are exactly the same as the ones chosen by **I-BT**. The following proposition immediately follows:

Proposition 4 **BI-BT** is sound and complete w.r.t the set of abstract solutions.

Fig. 3 shows the backtracking tree of **BI-BT** on Ex. 1. Refinement lists now contain bi-intervals and are smaller than in Fig. 2. The property required for FC is satisfied.

4.3. Extension to forward checking

BI-BT can be enhanced with FC (**BI-BT+FC**):

Proposition 5 Suppose $pre_V(v) = (v_1, \dots, v_k)$ and $\alpha(v_1), \dots, \alpha(v_i)$ are identical in two distinct branches of the search tree. Then $\lambda(v, v_1), \dots, \lambda(v, v_i)$ are also identical.

Proposition 6 **BI-BT+FC** is sound and complete w.r.t the set of abstract solutions.

5. Experiments

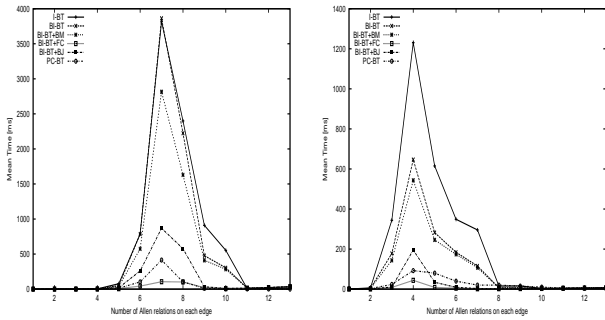
We compare the three algorithms (**I-BT**, **BI-BT**, **BI-BT+FC**) presented in this paper with the basic path-consistency-based one in [10] denoted **PC-BT**². Moreover, we also implement other optimizations schemes such as BackMark [6] (**BI-BT+BM**) that also relies on Prop. 5 and BackJump [6, 4, 11] (**BI-BT+BJ**)³.

Our benchmark was composed of 2600 complete and half-complete TCNs with 10 variables each. By varying the constraint hardness, we observe the efficiency of our algorithms in the transition region [14]. We ran the 6 above-mentioned algorithms on the instances of the benchmark using an Intel 2GHz machine with 2Gb of RAM. Note that the design of this benchmark (random instances covering the transition region) is meant to evaluate these algorithms on hard instances.

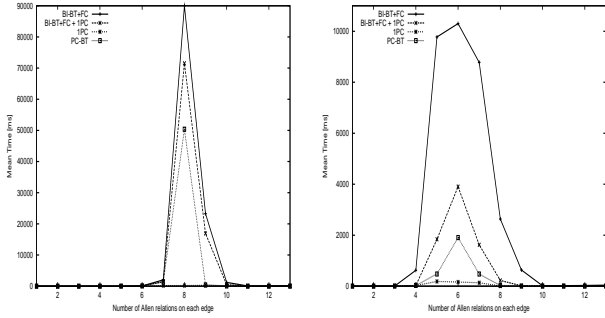
We have represented in Fig. 4(a) and 4(b) the average run time of each of the algorithms to check the satisfiability of the networks. Moreover, we apply on the same kind of TCNs with 15 variables each **BI-BT+FC** with a pre-step of path-consistency [1] (the cost of this elementary pre-step, that does not solve the problem by itself, is visualized by **IPC**). Results are reported in Fig. 4(c) and 4(d). Our interpretation of the results follows:

²We do not use the C-program of [10] to keep the same instantiation order as in **BI-BT**. Moreover, we do not want results to be twisted by additional optimizations: we only compare two look-ahead techniques.

³The Java-program that was used for the evaluation is available from <http://temporalsolver.gforge.inria.fr>



(a) 10 variables, 100% degree size. (b) 10 variables, 50% degree size.



(c) 15 variables, 100% degree size. (d) 15 variables, 50% degree size.

Figure 4. Experimental results.

- **BI-BT** is equivalent or better than **I-BT** due to the complex bi-interval refinements involved and to its larger partition;
- however, **BI-BT** can be improved: its extensions **BI-BT+BM** and **BI-BT+BJ** are better than **I-BT**, and its extension to **BI-BT+FC** is clearly more efficient;
- **PC-BT** filters more than **BI-BT+FC** (the search tree is on average 10 times smaller). However, **BI-BT+FC** can be improved in a hybrid algorithm that uses **PC-BT** because applying one step of path-consistency (**IPC**) before **BI-BT+FC** improves **BI-BT+FC**.

We wanted to compare our results with the efficient but incomplete local search-based algorithm **TSAT** [15]. We ran **BI-BT+FC** on random instances with 80 variables with the same hardness parameters as in [15]. On highly constrained problems, **BI-BT+FC** rapidly cuts all branches of the search tree, while **TSAT** has its worse results and is slower than **BI-BT+FC** (or **PC-BT**). While **BI-BT+FC** fails in the transition region, **TSAT** remains quick but misses most solutions.

6. Conclusion

We have presented sound and complete algorithms for satisfiability of TCNs. They do not rely on usual path-consistency techniques but use the weaker forward checking. We have shown that a partition of infinite domains in

bi-intervals allows backtracking optimizations. Moreover, these algorithms can be used in a hybrid algorithm that uses generic CSPs optimization techniques as well as specific TCNs path-consistency optimizations thus improving the backtracking part of path-consistency-based algorithms.

Before implementing this hybrid algorithm, we intend to explore other generic techniques such as variables and constraints orderings [10], structure-based optimizations [2], and use more efficient look-ahead filtering techniques such as MAC [3].

References

- [1] J. F. Allen. Maintaining knowledge about temporal intervals. *Com. of the ACM*, 26(11):832–843, 1983.
- [2] J.-F. Baget and Y. S. Tognetti. Backtracking through biconnected components of a constraint graph. In *Proc. of IJCAI'01*, pages 291–296, 2001.
- [3] C. Bessière and J.-C. Régin. MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems. In *Proc. of CP'96*, pages 61–75, 1996.
- [4] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41(3):273–312, 1990.
- [5] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
- [6] J. Gaschnig. Performance measurement and analysis of certain search algorithms. Technical Report CMU-CS-79-124, Carnegie Mellon University, 1979.
- [7] S. W. Golomb and L. D. Baumert. Backtrack programming. *Journal of the ACM*, 12(5):516–524, 1965.
- [8] R. M. Haralick and G. L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–314, 1980.
- [9] I. Meiri. Combining qualitative and quantitative constraints in temporal reasonings. *Artificial Intelligence*, 87:343–385, 1996.
- [10] B. Nebel. Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ORD-horn class. In *Proc. of ECAI'96*, pages 38–42, 1996.
- [11] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3):268–299, 1993.
- [12] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proc. of ECAI'90*, pages 550–556, 1990.
- [13] E. Schwalb and R. Dechter. Processing temporal constraint networks. *A. I.*, 93:29–61, 1997.
- [14] B. M. Smith and S. A. Grant. Sparse constraint graphs and exceptionally hard problems. In *Proc. of IJCAI'95*, pages 646–654, 1995.
- [15] J. Thornton, M. Beaumont, A. Sattar, and M. Maher. A local search approach to modelling and solving interval algebra problems. *J. of Logic and Computation*, 14(1):93–112, 2004.
- [16] 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.