ELSEVIER

Artificial Intelligence 93 (1997) 29-61

Artificial Intelligence

Processing disjunctions in temporal constraint networks ★

Eddie Schwalb *, Rina Dechter [1]

Department of Information and Computer Science, University of California at Irvine, CA 92717, USA

Received October 1995; revised December 1996

Abstract

Temporal constraint satisfaction problems (TCSPs) provide a formal framework for representing and processing temporal knowledge. Deciding the consistency of TCSPs is known to be intractable. We demonstrate that even local consistency algorithms like path-consistency (PC) can be exponential on TCSPs due to the fragmentation problem. We present two new polynomial approximation algorithms, Upper-Lower Tightening (ULT) and Loose Path-Consistency (LPC), which are efficient yet effective in detecting inconsistencies and reducing fragmentation. Our experiments on hard problems in the transition region show that LPC has the best effectiveness-efficiency tradeoff for processing TCSPs. When incorporated within backtrack search, LPC is capable of improving search performance by orders of magnitude. © 1997 Published by Elsevier Science B.V.

1. Introduction

Problems involving temporal constraints arise in various areas including temporal databases [6], diagnosis [13], scheduling [24,25], planning [19], common sense reasoning [28] and natural language understanding [2]. Among the formalisms for expressing and reasoning about temporal constraints are the interval algebra [1], point algebra [32], Temporal constraint satisfaction problems (TCSPs) [8] and models combining quantitative and qualitative constraints [14,20].

★ This work was partially supported by NSF grant IRI-9157636, by Air Force Office of Scientific Research grant AFOSR 900136 and by grants from TOSHIBA of America and Xerox.

* Corresponding author. E-mail: eschwalb@ics.uci.edu.

[1] E-mail: dechter@ics.uci.edu.

PII S0004-3702(97)00009-X

The two main types of Temporal Constraint Networks can be characterized as qualitative [1,32] and quantitative [8]. In the qualitative model, variables are time intervals or time points and the constraints are qualitative. In the quantitative model, variables represent time points and the constraints are metric. These two types have been combined into a single model [14,20]. In this paper we build upon the model proposed by Meiri [20], in which variables are either points or intervals and there are three types of constraints: metric point–point and qualitative point–interval and interval–interval.

Answering queries in constraint processing reduces to the tasks of determining consistency, computing a consistent scenario and computing the minimal network. When time is represented by (or isomorphic to the) integers[2], deciding consistency is NP-complete [8,20]. For qualitative networks, computing the minimal network is NP-hard [8,11]. In both qualitative and quantitative models, complexity stems from disjunctive relationships between pairs of variables and occur in many applications.

Example 1. A large NAVY cargo must leave New York starting on March 7, go through Chicago and arrive at Los Angeles within 8–10 days. From New York to Chicago the delivery requires 1–2 days by *air or* 10–11 days on the *ground*. From Chicago to Los Angeles the delivery requires 3–4 days by air *or* 13–15 days on the ground. In addition, we know that an AIRFORCE cargo needs to be transported using the same terminal in Chicago as required for the NAVY's cargo transportation (i.e. the intervals of NAVY and AIRFORCE shipments should not overlap). The transportation of the AIRFORCE cargo should start between March 17 and March 20 and requires 3–5 days by air *or* 7–9 days on the ground.

Given the above constraints, we are interested in answering questions such as: "are the constraints satisfiable?", "can the NAVY cargo arrive in Los Angeles on March 13–14?", "when should the cargo arrive in Chicago?", "how long may the NAVY cargo transportation take?". The first two queries reduce to deciding consistency and the third and fourth queries reduce to computing the minimal network.

Since answering such queries is inherently intractable, this paper focuses on the design of efficient and effective polynomial *approximation* algorithms for deciding consistency and computing the minimal network. The common approximation algorithm enforces path-consistency (PC) [8]. As we demonstrate, in contrast to discrete CSPs, enforcing path-consistency on quantitative TCSPs is exponential. This is because in the path-consistent quantitative TCSP intervals are broken into several smaller subintervals. This may result in an exponential blowup, leading to what we call fragmentation.

We present two algorithms for bounding fragmentation called Upper–Lower Tightening (ULT) and Loose Path-Consistency (LPC). We show that these algorithms avoid fragmentation and are effective in detecting inconsistencies. We also discuss five variants of the main algorithms, called ULT-2, Directional ULT (DULT), LPC-2, Directional LPC (DLPC) and Partial LPC (PLPC).

[2] This is always the case in practice.

We address two questions empirically:

(1) which of the algorithms presented is preferable for detecting inconsistencies, and
(2) how effective are the proposed algorithms when used to improve backtrack search by preprocessing and (guiding the search) by forward checking.

To answer the first question, we show that enforcing path-consistency may indeed be exponential in the number of intervals per constraint while ULT's execution time is almost constant. Nevertheless, ULT is able to detect inconsistency in about 70% of the cases in which PC does. Algorithm LPC further improves on ULT; it is both efficient and capable of detecting almost all of the inconsistencies detected by PC.

To answer the second question, we apply the new algorithms in three ways:

(1) in a preprocessing phase for reducing the fragmentation before initiating search,
(2) in a forward checking algorithm for reducing the fragmentation during the search and detecting dead-ends early, and
(3) in an advice generator for dynamic variable ordering.

Through experiments with hard problems which lie in the transition region (defined by [4,21]), we show that both ULT and LPC are preferred to PC and that LPC is the best algorithm overall. We conclude that the performance of backtrack search can be improved by several orders of magnitude when using LPC for preprocessing, forward checking and dynamic variable ordering.

The organization of the paper is as follows. Section 2 summarizes the model of TCSPs and the known algorithms for processing them. Section 3 presents algorithm Upper–Lower Tightening (ULT) and Section 4 presents a new tractable class based on ULT. Section 5 presents Loose Path-Consistency (LPC). Section 6 extends the results of Sections 3, 4 and 5 to networks of combined qualitative and quantitative constraints. Section 7 presents backtracking algorithms and Section 8 provides an empirical evaluation.

2. Temporal Constraint Networks

There are three kinds of temporal constraint satisfaction problems (TCSPs):

(1) *qualitative* TCSPs, widely known as Allen's interval algebra [1],
(2) *quantitative* TCSPs, introduced in [8], and
(3) *combined* qualitative and quantitative TCSPs, introduced in [20].

For simplicity of exposition, we will present our algorithms for the restricted model of quantitative TCSPs first. Thereafter, in Section 6, we extend these algorithms to process Meiri's combined model [20].

A quantitative TCSP involves a set of variables, $X_1, \ldots, X_n$, having *continuous* domains, each representing a time point. Each constraint C is a set of intervals

$$C \stackrel{\text{def}}{=} \{I_1, \ldots, I_n\} = \{[a_1, b_1], \ldots, [a_n, b_n]\}.$$

A unary constraint C_i restricts the domain of the variable X_i to the given set of intervals

$$C_i \stackrel{\text{def}}{=} (a_1 \leqslant X_i \leqslant b_1) \cup \cdots \cup (a_n \leqslant X_i \leqslant b_n).$$

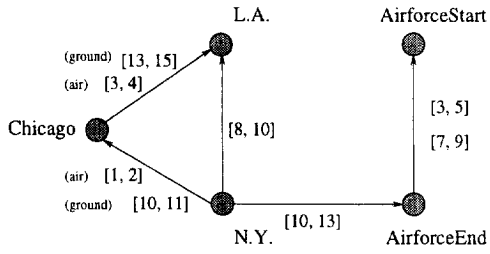


Fig. 1. The constraint graph for the metric portion of the logistics problem.

A binary constraint C_{ij} over X_i, X_j restricts the permissible values for the distance $X_j - X_i$; it represents the disjunction

$$C_{ij} \stackrel{\text{def}}{=} (a_1 \leqslant X_j - X_i \leqslant b_l) \cup \cdots \cup (a_n \leqslant X_j - X_i \leqslant b_n).$$

All intervals are assumed to be open and pairwise disjoint.

Example 2. Consider the cargo example given in the Introduction. Let the variables be:

X_0 = Jan 1, namely the beginning of the time line,

$X_{\text{N.Y.}}$ = time point at which the NAVY cargo was shipped out of N.Y.,

X_{Chicago} = time point at which the NAVY cargo arrived into and was shipped out of CHICAGO,

$X_{\text{L.A.}}$ = time point at which the cargo arrived into L.A.,

$X_{\text{AirforceStart}}$ = time point at which the AIRFORCE shipment *starts*,

$X_{\text{AirforceEnd}}$ = time point at which the AIRFORCE shipment *ends*.

The *metric* constraints are:

$X_{\text{N.Y.}} - X_0 \in [\text{March } 7, \text{March } 7]$,

$X_{\text{Chicago}} - X_{\text{N.Y.}} \in [1,2] \cup [10,11]$,

$X_{\text{L.A.}} - X_{\text{Chicago}} \in [3,4] \cup [13,15]$,

$X_{\text{L.A.}} - X_{\text{N.Y.}} \in [8,10]$,

$X_{\text{AirforceEnd}} - X_{\text{AirforceBegin}} \in [3,5] \cup [7,9]$,

$X_{\text{AirforceBegin}} - X_{\text{N.Y.}} \in [10,13]$.

Definition 3 (*Solution*). A tuple $X = (x_1, \ldots, x_n)$ is called a *solution* if the assignment $X_1 = x_1, \ldots, X_n = x_n$ satisfies all the constraints. The network is *consistent* iff at least one solution exists.

A quantitative TCSP can be represented by a *directed constraint graph*, where nodes represent variables and an edge $i \rightarrow j$ indicates that a constraint C_{ij} is specified. Every edge is labeled by the interval set as illustrated in Fig. 1. A special time point X_0 is

introduced to represent the "beginning of the world". All times can be specified relative to X_0 and thus each unary constraint C_i can be represented as a binary constraint C_{0i} (having the same interval representation). The constraint graph representing the logistics example is given in Fig. 1.

The minimal network is useful for answering a variety of queries, as described below, because it describes explicitly all the implicit (induced) binary constraints.

Definition 4 (*Minimal network*). A value v_i and v_{ij} is a *feasible value* of X_i and $X_j - X_i$, respectively, if there exists a solution in which $X_i = v$ and $X_j - X_i = v_{ij}$ respectively. The *minimal domain* of a variable is the set of all *feasible values* of that variable. A *minimal constraint* C_{ij} between X_i and X_j is the set of feasible values for $X_j - X_i$. A network is minimal iff its domains and constraints are minimal.

2.1. Answering queries

For completeness, we describe the set of queries that the quantitative TCSP model is designed to support. Consider the following sample queries:

(1) Is the network consistent, and if so, what is a possible scenario?
(2) *Can* X_i occur 5 to 10 minutes after X_j?
(3) *Must* X_i occur 5 to 10 minutes after X_j?
(4) At what possible times can event X_i occur?
(5) Given the time at which event X_i occurred, when can X_j occur?

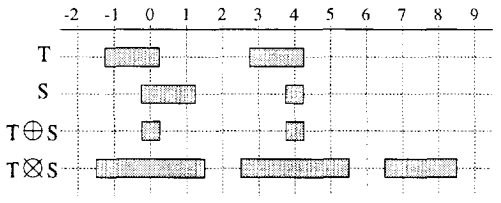
These queries can be partitioned into two groups: those that can be reduced to the task of deciding consistency and those that require computing the minimal network.

Clearly, Query 1 requires testing the consistency of the TCSP. To answer Query 2, we add the constraint $X_j - X_i \in [5, 10]$ and test for consistency. If the resulting network is consistent the answer to the query is *yes*; otherwise it is *no*. Query 3, often referred to as entailment, can be answered by adding (to the network) the negation of the constraint, namely $X_j - X_i \in [-\infty, 5) \cup (10, \infty]$, and checking for inconsistency. If consistency was detected by computing a solution, that solution provides a counter example that shows how X_i can occur less than 5 minutes or more than 10 minutes after X_j.

Queries 4 and 5 can be processed in constant time by a simple table lookup, after the equivalent minimal network (recall Definition 4) has been computed. The event associated with X_i can occur at time t for every $t \in C_{0i}$, where C_{0i} is the constraint between X_0 and X_i in the minimal network. Given that X_i occurs at time t_1, event X_j can occur at time $t_2 \in C_{ij} - t_1$, where C_{ij} is the constraint between X_i and X_j in the minimal network.

2.2. Path-consistency

Deciding whether a given network is consistent is NP-complete [8] and deciding whether it is minimal is NP-hard (which subsumes NP-complete). Therefore, it is common to use algorithms that detect some (but not all) inconsistencies and tighten the constraints to obtain an approximation of minimal constraints. Such algorithms enforce local k-consistency by ensuring that every subnetwork with k variables is minimal



$$T = \{[-1.25, 0.25],\ [2.75, 4.25]\}$$
$$S = \{[-0.25, 1.25],\ [3.77, 4.25]\}$$
$$T \cap S = \{[-0.25, 0.25],\ [3.75, 4.25]\}$$
$$T \odot S = \{[-1.50, 1.50],\ [2.50, 5.50],\ [6.50, 8.50]\}$$

Fig. 2. An illustration of the $\cap$ and the $\odot$ operations.

[7]. Here, we present path-consistency (3-consistency) for quantitative TCSPs. For qualitative TCSPs, 3,4-consistency algorithms are covered by [30].

Path-consistency is defined using the $\cap$ and the $\odot$ operations (see Fig. 2):

Definition 5 (*Operators*). Let $T = \{I_1, \ldots, I_l\}$ and $S = \{J_1, \ldots, J_m\}$ be two sets of intervals which can correspond to either unary or binary constraints.

(1) The *intersection* of T and S, denoted $T \cap S$, admits only values that are allowed by both of them.

(2) The *composition* of T and S, denoted $T \odot S$, admits only values r for which there exists $t \in T$ and $s \in S$ such that $r = t + s$ (Fig. 2).

The intuition behind enforcing path-consistency is as follows: We would like to compute the constraints induced by the composition of $C_{12} \odot C_{23} \odot \cdots \odot C_{k-1,k}$ along the path from X_1 to X_k. After path-consistency is enforced, we are guaranteed that $C_{1,k}$ is tighter than or equal to the constraint induced along this path.

Definition 6. A constraint C_{ij} is *path-consistent* iff $C_{ij} \subseteq \cap_{\forall k}(C_{ik} \odot C_{kj})$ and a network is *path-consistent* iff all its constraints are *path-consistent*.

Any arbitrary consistent quantitative TCSP with non-dense time domains (as is always the case in practice) can be converted into an equivalent *path-consistent* network by repeatedly applying the relaxation operation $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$ until a fixed point is reached. If the domains are dense, it is unclear under what conditions a fixed point can be reached in finite time. Fig. 3 presents an algorithm for enforcing path-consistency. For completeness, we also describe a weaker yet more efficient version of path-consistency, called Directional Path-Consistency (DPC), which is tied to a particular ordering of the variables [9].

Theorem 7 (Dechter et al. [8]). *If time is not dense then algorithms* PC *and* DPC *terminate in* $O(n^3R^3)$ *and* $O(n^3R^2)$ *steps respectively, where n is the number of variables and R is the range of the constraints, i.e. the difference between the lowest and highest numbers specified in the input network.*

Algorithm PC

1. $Q \leftarrow \{(i,k,j) | (i<j)\ and\ (k \neq i,j)\}$
2. **while** $Q \neq \{\}$ **do**
3. select and delete a path (i,k,j) from Q
4. **if** $C_{ij} \neq C_{ik} \odot C_{kj}$ **then**
5. $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$
6. **if** $C_{ij} = \{\}$ **then** exit (inconsistency)
7. $Q \leftarrow Q \cup \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\}$
8. **end-if**
9. **end-while**

Algorithm DPC

1. **for** $k \leftarrow n$ downto 1 by -1 **do**
2. **for** $\forall i,j < k$ such that $(i,k),(k,j) \in E$ **do**
3. **if** $C_{ij} \neq C_{ik} \odot C_{kj}$ **then**
4. $E \leftarrow E \cup (i,j)$
5. $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$
6. **if** $C_{ij} = \{\}$ **then** exit (inconsistency)
7. **end-if**
8. **end-for**
9. **end-for**

Fig. 3. Algorithms PC and DPC.

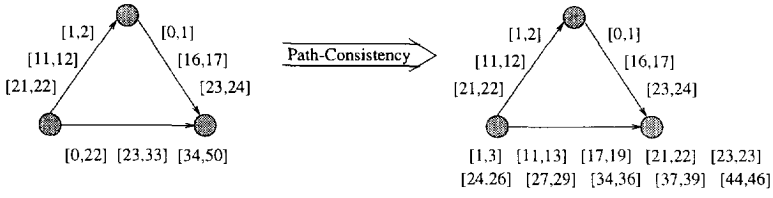


Fig. 4. The fragmentation problem.

Example 8. Consider a constraint $X_j - X_i \in [-1000,-990] \cup [-800,+800] \cup [990,1000]$. The range R of this constraint is $[-1000,1000]$. For such R, the bound given in Theorem 7 suggests that PC might need to update the constraints thousands of times.

2.3. Fragmentation

In contrast to discrete CSPs, enforcing path-consistency on quantitative TCSPs is problematic when the range R is large or the domains are continuous [8,24]. An upper bound on the number of intervals in $T \odot S$ is $|T| \cdot |S|$, where $|T|$ and $|S|$ are the number of intervals in T and S respectively. As a result, the total number of intervals in the path-consistent network might be exponential in the number of intervals per constraint in the input network, yet bounded by R when integer domains are used.

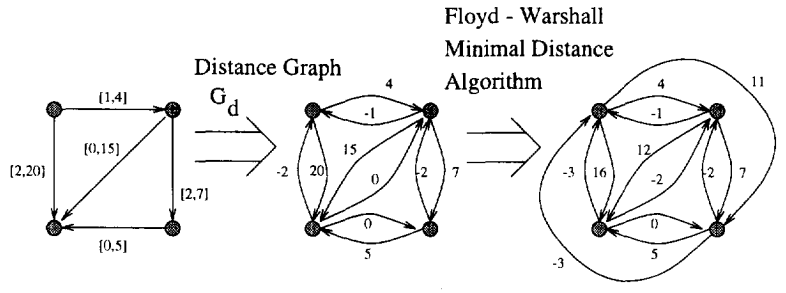


Fig. 5. Processing an STP.

Example 9. Consider the network presented in Fig. 4, having three variables, three constraints and three intervals per constraint. After enforcing path-consistency, two constraints remain unchanged in the path-consistent network while the third is broken into 10 subintervals. As this behavior is repeated over numerous triangles in the network, the number of intervals may grow exponential in the number of intervals per constraint.

3. Upper–Lower Tightening (ULT)

Enforcing path-consistency computes a tighter equivalent network that approximates the minimal network and is useful for answering a variety of queries. The problem with enforcing path-consistency is that the relaxation operation $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$ may increase the number of intervals in C_{ij}. Our idea is to compute looser constraints which consist of fewer intervals that subsume all the intervals of the path-induced constraint.

3.1. Simple temporal problems

Fragmentation does not occur when we enforce path-consistency on the special class of quantitative TCSPs called the *Simple Temporal Problem* (STP). In these networks, only a single interval is specified per constraint.

An STP can be associated with a directed edge-weighted graph, G_d, called a *distance graph* (d-graph), having the same vertices as the constraint graph G; each edge $i \rightarrow j$ is labeled by a weight w_{ij} representing the constraint $X_j - X_i \leqslant w_{ij}$, as illustrated in Fig. 5. An STP is consistent iff the corresponding d-graph G_d has no negative cycles and the minimal network of the STP corresponds to the *minimal distances* in G_d. Therefore, an all-pairs shortest path procedure (Fig. 5) is equivalent to enforcing path-consistency and is complete for STPs [8].

3.2. Avoiding fragmentation

The algorithm for approximating path-consistency, called Upper–Lower Tightening (ULT), utilizes the fact that an STP is tractable. The algorithm treats the extreme points

Algorithm Upper-Lower Tightening (ULT)

1. **input:** N
2. $N''' \leftarrow N$
3. **repeat**
4. $\quad N \leftarrow N'''$
5. $\quad$ compute N', N'', N'''.
6. **until** $\forall ij \quad (low(C_{ij}''') = low(C_{ij}))$ *and* $(high(C_{ij}''') = high(C_{ij}))$
   ; which implies no change
   or $\exists ij \quad (high(C_{ij}''') < low(C_{ij}'''))$
7. **if** $\forall ij \quad (high(C_{ij}''') > low(C_{ij}'''))$ **output:** N'''
   otherwise **output:** "Inconsistent."

Fig. 6. The Upper-Lower Tightening (ULT) algorithm.

of all the intervals associated with a single constraint as one big interval, yielding an STP, and then performs path-consistency on that STP. This process cannot increase the number of intervals per constraint. Finally, we intersect the resulting simple path-consistent minimal network with the input network.

Definition 10 (*Upper-Lower Tightening*). Let $C_{ij} = [I_1, \ldots, I_m]$ be the constraint over variables X_i and X_j and let $low(C_{ij})$ and $high(C_{ij})$ be the lower and upper bounds of C_{ij}, respectively. We define N', N'', and N''' as follows:

- N' is an STP derived from N by relaxing its constraints to

$$C'_{ij} = [low(C_{ij}), high(C_{ij})].$$

- N'' is the minimal network of N' (N' is an STP).
- N''' is the intersection of N'' and N, namely $C_{ij}''' = C_{ij}'' \cap C_{ij}$.

Algorithm Upper-Lower Tightening (ULT) is presented in Fig. 6. The network N' is a relaxation of N. N'' is computed by applying the all-pairs shortest path algorithm on N'. Because N'' is equivalent to N', intersecting N'' with N results in a network that is equivalent to N.

Lemma 11. *Let N be the input to* ULT *and R be its output.*

(1) *The networks N and R are equivalent.*

(2) *Every iteration of* ULT (*except the last one*) *removes at least one interval.*

Proof. (1) Let $Sol(N)$ denote the set of solutions of the network N, then $Sol(N) \subseteq Sol(N') = Sol(N'')$. This implies that $Sol(N) \cap Sol(N'') = Sol(N)$ and therefore $Sol(N''') = Sol(N)$.

(2) Let N'_i and N''_i be the networks N' and N'' at iteration i. If an interval is not removed at iteration i, $N''_i = N'_{i+1} = N''_{i+1}$, which implies no change. □

Algorithm ULT computes looser networks than those resulting from enforcing full path-consistency. A qualitative worst-case comparison is given in Section 5.1 and is depicted in Fig. 14. A quantitative empirical comparison is given in Section 8.

Example 12. An example run of ULT on a sample problem instance is given in Fig. 7. We start with N and compute $N'_{(1)}$, $N''_{(1)}$ and $N'''_{(1)}$. Thereafter, we perform the second iteration in which we compute $N'_{(2)}$, $N''_{(2)}$ and $N'''_{(2)}$ and finally, in the third iteration, there is no change. The first iteration removes two intervals, while the second iteration removes one. In addition, ULT computes an induced constraint C_{02}, which allows inferring a new implicit fact that was not specified explicitly in the input network.

Theorem 13. *Algorithm* ULT *terminates in* $O(n^3ek+e^2k^2)$ *steps where n is the number of variables, e is the number of edges, and k is the maximal number of intervals in each constraint.*

Proof. Because computing N' requires processing every interval in the network at most once, this computation requires $O(ek)$ steps. Computing N'' from N' can be done by applying the all-pairs shortest path algorithm (e.g. Floyd–Warshall) and thus requires $O(n^3)$ steps. Computing the intersection $T \cap S$ of two *sorted* constraints requires $O(|T|+|S|)$ steps, thus computing N''' from N'' requires $O(ek)$ steps. This means that each iteration requires $O(n^3+ek)$ steps. The halting condition (Fig. 6, line 6) implies that at every iteration at least one interval must be removed (Lemma 11). Therefore, at most $O(ek)$ iterations are performed yielding a total complexity of $O(n^3ek+e^2k^2)$ steps. □

To explain the difference between ULT and PC, we view every disjunctive constraint as a single interval with *holes*. The single interval specifies the upper and lower bounds of legal values while the holes specify intervals of illegal values.

Lemma 14. *Algorithms* ULT *and* PC *compute the same upper and lower bounds.*

Proof. The lower and upper bounds are modified using the $\cap$ and the $\odot$ operators. We observe that $low(C_{ik} \odot C_{kj}) = low(C_{ik}) + low(C_{kj})$ which is equal to the lower bound of $[low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})]$. A similar observation is made for the upper bound. Consequently, the lower and upper bounds of $C_{ij} \cap ([low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})])$ and $C_{ij} \cap (C_{ik} \odot C_{kj})$ are equal. Additional iterations performed by PC only enlarge the "holes". □

Thus, the difference between ULT and PC is the propagation of the holes. In contrast to PC, ULT is guaranteed to converge in $O(ek)$ iterations even if the interval boundaries are not rational numbers.

3.3. *Variations of* ULT

While an iteration of ULT is divided into three sequential stages that involve the whole network, algorithm PC uses simpler local operations over triplets of variables and admits parallel execution. We next present two variations on ULT, called ULT-2 and

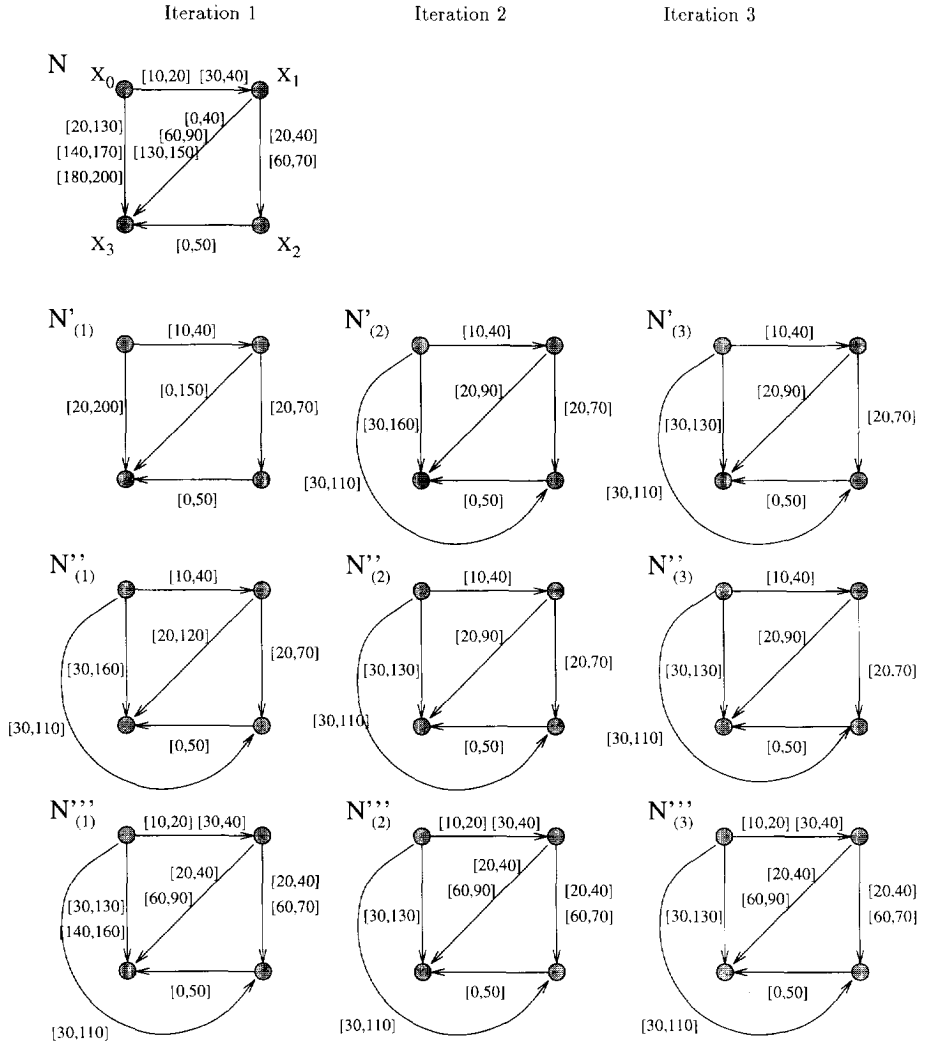


Fig. 7. A sample run of ULT.

Directional ULT (DULT), which perform such local computations (see Fig. 8). We use $low(C_{ij})$ and $high(C_{ij})$ to denote, respectively, the lowest lower bound and highest upper bound of the union of the intervals in C_{ij}.

Theorem 15. *Given a network N, let n be the number of variables, e the number of constraints and k the maximum number of intervals per constraint.*

(1) *Algorithms* ULT-2 *and* DULT *terminate in* $O(n^3k^2 + ek^3n)$ *and* $O(n^3k^2)$ *steps respectively and compute a network equivalent to their input network.*

(2) *Algorithm* ULT-2 *computes a tighter network than* DULT.

Algorithm ULT-2

1. $Q \leftarrow \{(i,k,j) | (i<j)\ and\ (k \neq i,j)\}$
2. **while** $Q \neq \{\}$ **do**
3. select and delete a path (i,k,j) from Q
4. $T''_{ij} \leftarrow C_{ij} \cap ([low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})])$
5. **if** $T'_{ij} = \{\}$ **then** exit (inconsistency)
6. **if** $T''_{ij} \neq C_{ij}$ **then**
$Q \leftarrow Q \cup \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\}$
7. $C_{ij} \leftarrow T''_{ij}$
8. **end-while**

Algorithm DULT

1. **for** $k \leftarrow n$ downto 1 by -1 **do**
2. **for** $\forall i,j < k$ such that $(i,k),(k,j) \in E$ **do**
3. $T''_{ij} \leftarrow C_{ij} \cap ([low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})])$
4. **if** $T''_{ij} = \{\}$ **then** exit (inconsistency)
5. **if** $T''_{ij} \neq C_{ij}$ **then** $E \leftarrow E \cup (i,j)$
6. $C_{ij} \leftarrow T''_{ij}$
7. **end-for**
8. **end-for**

Fig. 8. Algorithms ULT-2 and DULT.

Proof. (1) Algorithm ULT-2 initializes the queue with $O(n^3)$ triangles. A set of $O(n)$ triangles is added to Q (Fig. 8, Algorithm ULT-2, line 6) only if at least one interval was removed from the network, and therefore at most $O(ekn)$ triangles are added. Since computing $T \odot S$ requires at most $O(k^2)$ steps, the total complexity for ULT-2 is $O(n^3k^2 + ek^3n)$. Algorithm DULT performs a single pass of $O(n^3)$ triangles and each triangle requires $O(k^2)$ steps.

(2) Every triangle that is processed in DULT is also processed in ULT-2 but not vice versa, thus DULT is weaker. □

Algorithm ULT can be extended to process discrete constraint satisfaction problems (see Appendix A).

4. The STAR tractable class

This section analyzes the class of quantitative TCSPs in which the binary constraints C_{ij} specify single intervals but the unary constraints C_{0i} may specify an arbitrary number of intervals. It subsumes the class of convex point algebra networks with holes in their domains [20], but it is not comparable to the class of STPs with disjunctions of inequations [15] over dense domains.

Definition 16 (*STP upper bound*). The STP upper bound of a network N, denoted $N' = STP(N)$ is such that $C'_{ij} = [low(C_{ij}), high(C_{ij})]$.

Lemma 17 (Dechter et al. [8]). *For a minimal STP with constraints* $C_{ij} = [L_{ij}, U_{ij}]$, *the instantiation* $X_i = L_{0i}$ *is a solution.*

Lemma 18. *For every quantitative TCSP N, if STP(N) is minimal, then the instantiation* $X_i = low(C_{0i})$ *is a solution of N if all the binary constraints (i.e.* C_{ij}, $\forall i > 0$, $\forall j > 0$) *specify a single interval.*

Proof. From Lemma 17 it follows that this instantiation is a solution of $STP(N)$. Thus, all the binary constraints are satisfied by this instantiation. Clearly, because $L_{0i} \in C_{0i} = C_i$, all the disjunctive constraints C_{0i} are also satisfied by this instantiation. □

Lemma 19. *Algorithm* ULT *computes a network N whose STP(N) is minimal.*

Proof. Follows immediately from Definition 10. □

From Lemmas 18 and 19 we obtain the following theorem.

Theorem 20. *Algorithm* ULT *correctly decides consistency of TCSPs in which* C_{ij}, $\forall i > 0$, $\forall j > 0$, *is specified by a single interval.*

This class of problems is frequently encountered. Consider, for example, scheduling tasks that use resources available in a set of time windows. The availability of resources constrains the times at which tasks can be accomplished and can be describe by unary constraints. For example, suppose we would like to transport cargo from the east coast to the west coast. To use an air carrier we need to consider resource availability constraints described by disjunctive unary constraints on the times that cargo loading and unloading can occur.

This class can be generalized using the notion of a *disjunctive constraint graph* $G(V, E)$ whose vertices V correspond to variables and edges E specify disjunctive constraints only.

Corollary 21. ULT *is complete for TCSPs whose disjunctive constraint graph is a* STAR, *namely a tree in which all edges are incident on a single node.*

Proof. Label the root of the STAR by X_0 and apply Theorem 20. □

Moreover, even if the input TCSP is not a STAR, ULT may remove disjunctions and obtain a STAR network. In such cases, ULT is complete.

An important consequence of the above is that ULT reduces the search space by an exponential factor. Since the search algorithm need not consider all the disjunctive constraints connected to the node with the maximal degree, the search space is reduced by a factor of $O(k^{d(G)})$, where G is the *disjunctive constraint graph*, $d(G)$ is the maximal degree of G and k is the disjunction size, namely the number of intervals in each constraint. In Section 8.2.1 we show empirically that without preprocessing with ULT even tiny problems were computationally prohibitive and could not be solved in a reasonable amount of time.

Algorithm Loose Path-Consistency (LPC)

1. **input:** N
2. $N'' \leftarrow N$
3. **repeat**
4. $\quad N \leftarrow N''$
5. $\quad$ Compute N' by assigning $T'_{ij} = \cap_{\forall k}(C_{ik} \odot C_{kj})$, for all i, j.
6. $\quad$ Compute N'' by loosely intersecting $T''_{ij} = C_{ij} \lhd T'_{ij}$, for all i, j.
7. **until** $\exists i, j \ \ (T''_{ij} = \phi)$; inconsistency, or
   or $\forall i, j \ \ |T''_{ij}| = |C_{ij}|$; no interval removed.
8. **if** $\exists i, j \ \ (T''_{ij} = \phi)$ **then output** "inconsistent."
   else output: N''.

Fig. 9. The Loose Path-Consistency (LPC) algorithm.

Unfortunately, our analysis cannot be extended to networks whose disjunctive constraint graph is a general tree. Consider a triangle X_i, X_j, X_k with the constraint bounds $[L_{ij}, U_{ij}]$, $[L_{ik}, U_{ik}]$ and $[L_{kj}, U_{kj}]$ respectively. When $STP(N)$ is minimal we are guaranteed that $L_{ij} \geqslant L_{ik} + L_{kj}$. Thus, instantiating $X_k = X_i + L_{ik}$ and $X_j = X_k + L_{kj}$ does not guarantee that $X_j - X_i \in [L_{ij}, U_{ij}]$.

5. Loose Path-Consistency (LPC)

Now we present algorithm *Loose Path-Consistency* (LPC), which is stronger than ULT and its variants, namely it generates tighter approximations to PC. The algorithm is based on the following loose intersection operator.

Definition 22. Let $T = \{I_1, I_2, \ldots, I_r\}$ and $S = \{J_1, J_2, \ldots, J_s\}$ be two constraints. The *loose intersection*, $T \lhd S$ consists of the intervals $\{I'_1, \ldots, I'_r\}$ such that $\forall i \ I'_i = [L_i, U_i]$ where $[L_i, U_i]$ are the lower and upper bounds of the intersection $I_i \cap S$.

It is easy to see that the number of intervals in C_{ij} is not increased by the operation $C_{ij} \leftarrow C_{ij} \lhd (C_{ik} \odot C_{kj})$. In addition, $\forall k \ C_{ij} \supseteq C_{ij} \lhd (C_{ik} \odot C_{kj}) \supseteq C_{ij} \cap (C_{ik} \odot C_{kj})$ and $T \lhd S \neq S \lhd T$.

Example 23. Let $T = \{[1,4], [10,15]\}$ and $S = \{[3,11], [14,19]\}$. Then $T \lhd S = \{[3,4], [10,15]\}$, $S \lhd T = \{[3,11], [14,15]\}$ while $S \cap T = \{[3,4], [10,11], [14,15]\}$.

According to Definition 6, a constraint C_{ij} is path-consistent iff $C_{ij} \subseteq \cap_{\forall k}(C_{ik} \odot C_{kj})$. By replacing the intersection operator $\cap$ with the loose intersection operator $\lhd$, we can bound the fragmentation.

Algorithm LPC is presented in Fig. 9. The network N' is a relaxation of N and therefore loosely intersecting N'' with N results in an equivalent network.

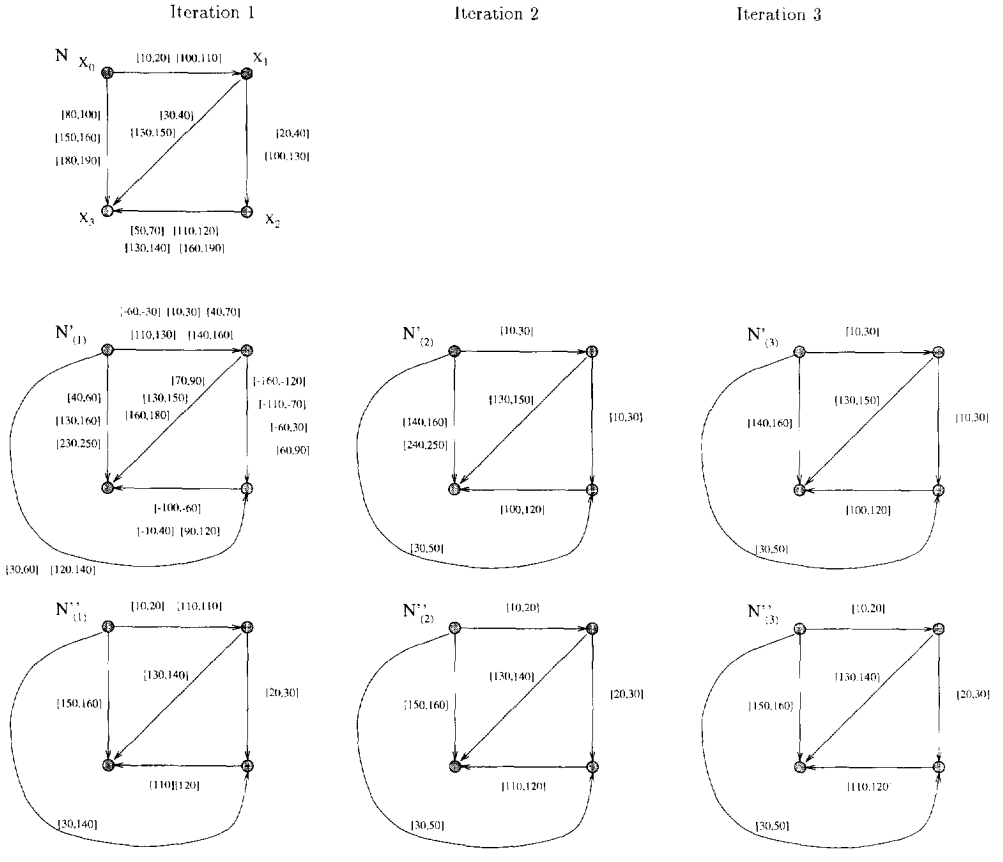


Fig. 10. A sample run of LPC.

Example 24. In Fig. 10 we show a trace of LPC on a sample quantitative TCSP. We start with N and compute $N'_{(1)}$ and $N''_{(1)}$. Thereafter, we perform a second iteration in which we compute $N'_{(2)}$ and $N''_{(2)}$. Finally, in the third iteration, there is no change. The first iteration removes seven intervals while the second iteration removes a single interval. We see that LPC explicates an induced constraint C_{02}, which allows to infer a new implicit fact about the times that event X_2 can occur. Note that applying ULT on the same network will have no effect and applying PC on it results in the same network as results from applying LPC.

Lemma 25. *Let N be the input to* LPC *and R be its output.*

(1) *The networks N and R are equivalent.*

(2) *Every iteration of* LPC *(excluding the last) removes at least one interval from one of the constraints.*

Proof. Immediate. □

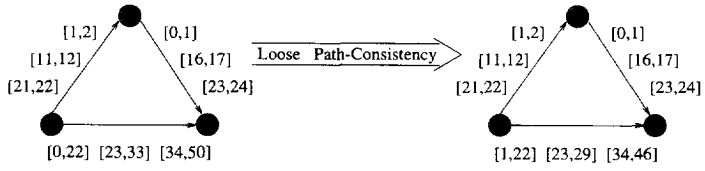


Fig. 11. Solving the fragmentation problem.

Theorem 26. *Algorithm* LPC *terminates in* $O(n^3k^3e)$ *steps where n is the number of variables, e is the number of constraints and k is the maximal number of intervals in each constraint.*

Proof. Computing N' requires processing every triangle in the network once, thus requires $O(n^3k^2)$ steps. Because in every iteration at least one interval is removed, there are at most ek iterations. The complexity is therefore $O(n^3k^3e)$. □

Algorithm LPC computes tighter networks than ULT. For detailed execution, see Fig. 10. To clarify the differences among ULT, LPC and PC, we can view every disjunctive constraint as a single interval with *holes* (as in Section 3.2). The single interval specifies the upper and lower bounds of legal values, while the holes specify intervals of illegal values.

Lemma 27. *Algorithms* ULT, LPC *and* PC *compute the same upper and lower bounds.*

Proof. Using the same arguments as in the proof of Lemma 14 we show that the lower and upper bounds of $C_{ij} \lhd (C_{ik} \odot C_{kj})$ and

$$C_{ij} \cap [low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})]$$

are equal to the bounds of $C_{ij} \cap (C_{ik} \odot C_{kj})$. □

Thus, the difference among ULT, LPC and PC is in their propagation of holes. Algorithm ULT does not change the holes. LPC may enlarge the holes, while PC may increase their number.

5.1. *Variations of* LPC

We next present two variations on LPC which have the same structure as PC and DPC. These algorithms, presented in Fig. 12, are called LPC-2 and Directional LPC (DLPC). They differ from PC and DPC only in using the loose intersection operator $\lhd$ instead of the strict intersection operator $\cap$.

Theorem 28. *Given a network N, let n be the number of variables, e be the number of constraints and k be the maximum number of intervals per constraint. Algorithms* LPC-2 *and* DLPC *terminate in* $O(n^3k^2 + ek^3n))$ *and* $O(n^3k^2)$ *steps, respectively, and they compute TCSPs which are equivalent to their input.*

Algorithm LPC-2

1. $Q \leftarrow \{(i,k,j) | (i<j)\ and\ (k \neq i,j)\}$
2. **while** $Q \neq \{\}$ **do**
3. select and delete a path (i,k,j) from Q
4. $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$
5. **if** $T'_{ij} = \{\}$ **then** exit (inconsistency)
6. **if** $|T'_{ij}| < |C_{ij}|$ **then**
$Q \leftarrow Q \cup \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\}$
7. $C_{ij} \leftarrow T'_{ij}$
8. **end-while**

Algorithm DLPC

1. **for** $k \leftarrow n$ downto 1 by -1 **do**
2. **for** $\forall i,j < k$ such that $(i,k),(k,j) \in E$ **do**
3. $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$
4. **if** $T'_{ij} = \{\}$ **then** exit (inconsistency)
5. **if** $|T'_{ij}| < |C_{ij}|$ **then** $E \leftarrow E \cup (i,j)$
6. $C_{ij} \leftarrow T'_{ij}$
7. **end-for**
8. **end-for**

Fig. 12. Algorithms LPC-2 and DLPC.

Proof. Algorithm LPC-2 applies the operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$ which does not change the set of solutions, and thus the resulting network is equivalent. Initially, the queue Q consists of $O(n^3)$ triangles. A set of $O(n)$ triangles is added to Q (LPC-2, line 6) only if at least one interval was removed from the network, and therefore at most $O(ekn)$ triangles are added. Since computing $T \odot S$ requires at most $O(k^2)$ steps the total complexity of LPC-2 is $O(n^3k^2 + ek^3n)$. Algorithm DLPC applies the operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$ at most $O(n^3)$ times. Each such operation does not change the set of solutions and requires $O(k^2)$ steps. Thus the overall complexity of DLPC is $O(n^3k^2)$. □

5.2. *Partial* LPC (PLPC)

To refine the tradeoff between effectiveness and efficiency, we suggest another variant for constraint propagation, called *Partial LPC* (PLPC). We apply the relaxation operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$ only in cases where C_{ij} and at least one of C_{ik} and C_{kj} is non-universal in the input network. Consider, for example, the tree network in Fig. 13(a) and the circle network in Fig. 13(b). The dashed lines outline several triangles that are not processed.

5.3. *Relative effectiveness*

The partial order on the effectiveness of all the algorithms presented in this paper is shown in Fig. 14. A directed edge from algorithm $\mathcal{A}_1$ to $\mathcal{A}_2$ indicates that $\mathcal{A}_2$ computes

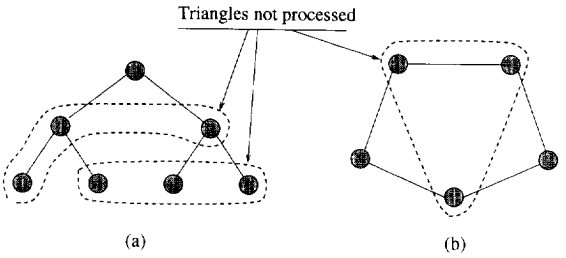


Fig. 13. The utility of PLPC.

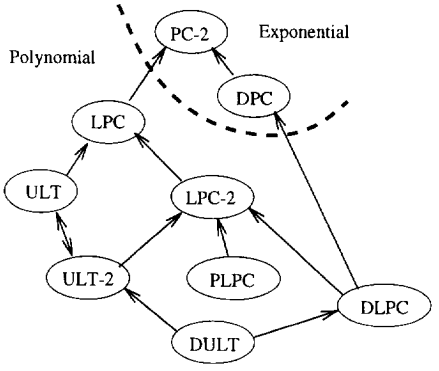


Fig. 14. The partial order on the effectiveness.

an equivalent network which is equal or tighter than $\mathcal{A}_1$ on an *instance by instance* basis. This means that $\mathcal{A}_2$ can detect inconsistencies that $\mathcal{A}_1$ cannot detect, but not vice versa. Note that algorithms PC and DPC are exponential.

6. Combining quantitative and qualitative constraints

In this section, we present Meiri's extension [20] which combines qualitative and metric constraints over time points and intervals.

A combined qualitative and quantitative TCSP involves a set of variables and a set of binary constraints over pairs of variables. There are two types of variables, point and interval variables. The constraint C_{ij} between a pair of variables, X_i, X_j is described by specifying a set of allowed relations, namely

$$C_{ij} \stackrel{\text{def}}{=} (X_i\ r_1\ X_j) \vee \cdots \vee (X_i\ r_k X_j). \tag{1}$$

There are three types of relations, or alternatively, disjunctive constraints:

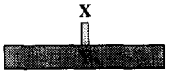
Relation	Symbol	Inverse	Example
X before Y	b	bi	
X starts Y	s	si	
X during Y	d	di	
X finishes Y	f	fi	
X after Y	a	ai	

Fig. 15. The five qualitative point–interval relations (X is a point and Y is an interval).

Relation	Symbol	Inverse	Example
X before Y	b	bi	
X equal Y	=	=	
X meets Y	m	mi	
X overlaps Y	o	oi	
X during Y	d	di	
X starts Y	s	si	
X finishes Y	f	fi	

Fig. 16. The 13 qualitative interval–interval relations.

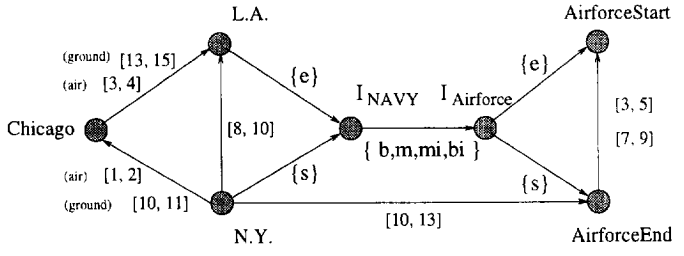


Fig. 17. The complete constraint graph of the logistics problem.

(1) A point–point constraint between two point variables X_i, X_j is *quantitative*[3] and has the form

$$X_j - X_i \in I_1 \cup \cdots \cup I_k,$$

where $I_1, \ldots, I_k$ are intervals.

(2) A point–interval constraint between a point variable and an interval variable, is *qualitative*, and is in the set {**before**, **starts**, **during**, **finishes**, **after**} abbreviated {b, s, d, f, a}, respectively (see Fig. 15) [20].

(3) An interval–interval constraint between two interval variables is *qualitative*, and is in the set

{**before**, **after**, **meets**, **met-by**, **overlaps**, **overlapped-by**, **during**, **contains**, **equals**, **starts**, **started-by**, **finishes**, **finished-by**},

abbreviated {b, bi, m, mi, o, oi, d, di, =, s, si, f, fi}, respectively (see Fig. 16) [1].

Example 29. Consider the cargo example of Section 1. Let the variables be:

X_0 = Jan 1, namely the beginning of the time line,

$X_{\text{N.Y.}}$ = time point at which the NAVY cargo was shipped out of N.Y.,

X_{Chicago} = time point at which the NAVY cargo arrived into and was shipped out of CHICAGO,

$X_{\text{L.A.}}$ = time point at which the cargo arrived into L.A.,

I_{NAVY} = transportation interval of the NAVY cargo.

I_{Airforce} = transportation interval of the AIRFORCE cargo.

$X_{\text{AirforceStart}}$ = time point at which the AIRFORCE shipment *starts*,

$X_{\text{AirforceEnd}}$ = time point at which the AIRFORCE shipment *ends*.

[3] In [20] a distinction is made between qualitative and quantitative point–point constraints.

The constraints are:

$X_{\text{N.Y.}} - X_0 \in [\text{March } 7, \text{March } 7]$,
$X_{\text{Chicago}} - X_{\text{N.Y.}} \in [1,2] \cup [10,11]$,
$X_{\text{L.A.}} - X_{\text{Chicago}} \in [3,4] \cup [13,15]$,
$X_{\text{L.A.}} - X_{\text{N.Y.}} \in [8,10]$,
$X_{\text{N.Y.}}\{\text{starts}\}I_{\text{NAVY}}$,
$X_{\text{L.A.}}\{\text{ends}\}I_{\text{NAVY}}$,
$X_{\text{AirforceBegin}}\{\text{starts}\}I_{\text{Airforce}}$,
$X_{\text{AirforceEnd}}\{\text{ends}\}I_{\text{Airforce}}$,
$X_{\text{AirforceEnd}} - X_{\text{AirforceBegin}} \in [3,5] \cup [7,9]$,
$X_{\text{AirforceBegin}} - X_{\text{N.Y.}} \in [10,13]$,
$I_{\text{NAVY}}\{\text{before}, \text{meets}, \text{met-by}, \text{after}\}I_{\text{Airforce}}$.

The last constraint means that I_{NAVY} and I_{Airforce} are disjoint. The constraint graph representing this network is given in Fig. 17.

6.1. *Extending* LPC *for combined networks*

For brevity we will describe the extension for LPC, but ULT can be extended using the same methodology. As defined in Section 2, the combined model involves three types of constraints: point–point (quantitative), point–interval (qualitative) and interval–interval (qualitative). Each node in a triangle can be either a point or an interval variable, resulting in $2^3 = 8$ types of triangles. We therefore modify the semantics of the $\triangleleft$ and the $\odot$ operators to accommodate all 8 types.

Let C_{ij}, C_{ik} and C_{kj} be the constraints on the pairs X_i, X_j, X_i, X_k and X_j, X_k. For computing $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \odot C_{kj})$, we use Meiri's tables except when quantitative constraints are used. We consider the following five cases:

Case 1: If X_i, X_j and X_k are interval variables, then Allen's transitivity table [1] is used to compute $C_{ik} \odot C_{kj}$ and the $\triangleleft$ operator is interpreted as the usual intersection operator.

Case 2: If both X_i and X_j are interval variables and X_k is a point variable, then Meiri's transitivity tables [20] are used to compute $C_{ik} \odot C_{kj}$ and the $\triangleleft$ operator is interpreted as the usual intersection.

Case 3: If exactly one of X_i and X_j is an interval variable and X_k is a point variable, then the quantitative point–point constraint, C_{ik} or C_{kj}, is translated into a qualitative point–point constraint (using $<$, $>$, $=$) and Meiri's transitivity tables [20] are used to compute $C_{ik} \odot C_{kj}$; the $\triangleleft$ operator is interpreted as the usual intersection.

Case 4: If X_i and X_j are point variables and X_k is an interval variable, then $C_{ik} \odot C_{kj}$ is computed using Meiri's transitivity tables [20]. If $C_{ik} \odot C_{kj} \neq \{<, >\}$ then the resulting constraint is translated into a single interval and the $\triangleleft$ operator is interpreted as the $\cap$ operator in Definition 3. Otherwise, to avoid increasing the number of intervals in C_{ij}, we set $T'_{ij} \leftarrow C_{ij}$ (i.e. no change).

Case 5: If X_i, X_j and X_k are point variables, then the composition operation used is described by Definition 3 and the $\triangleleft$ operator is described in Definition 5.

With these new definitions of the operators $\odot$ and $\triangleleft$, we can apply algorithms LPC, LPC-2 and DLPC for processing combined networks.

7. General backtracking

Algorithms ULT and LPC are useful for detecting inconsistencies and for explicating constraints, however they are not designed to find a consistent scenario (i.e. a solution). A brute-force algorithm for determining consistency or for computing consistent scenarios can decompose the network into separate simple subnetworks by selecting a single interval from each quantitative constraint and a single relation from a qualitative constraint [8,20]. Each subnetwork can then be solved separately in polynomial time by enforcing path-consistency, and the solutions can be combined. Alternatively, a naive backtracking algorithm can successively select one interval or relation from each disjunctive constraint as long as the resulting network is consistent [8,20]. Once inconsistency is detected, the algorithm backtracks. This algorithm can be improved by performing *forward checking* to reduce the number of possible future interval assignments during the labeling process.

Definition 30 (Meiri [20]). A *basic label* of an arc $i \rightarrow j$ is either a selection of a single interval from the interval set C_{ij} for quantitative constraints, or a selection of a single relation for qualitative constraints. A *singleton labeling* of N is a selection of a basic label for *all* the constraints in N and a *partial labeling* of N is a selection such that *some* constraints are assigned basic labels.

A singleton labeling of a combined network can be described by an STP [20]. Thus, deciding the consistency of a singleton labeling can be done in $O(n^3)$ steps, by enforcing path-consistency [20].

Lemma 31. *Algorithms* ULT, ULT-2, DULT, LPC, LPC-2 *and* DLPC *and their extension for processing combined networks decide consistency of a singleton labeling.*

Proof. When there are no disjunctions, the quantitative TCSP can be described by an STP, for which all of the above algorithms are complete. Enforcing path-consistency of a qualitative TCSP with no disjunctions is known to decide its consistency [1,20]. □

We can apply backtrack search with forward checking in the space of partial labelings as follows: The algorithm chooses a disjunctive constraint and replaces it with a single interval (if metric) or a single relation (if qualitative) from that constraint. When the constraints are chosen in a dynamic order, the constraint with the smallest disjunction size is selected for labeling. Thereafter, the network can be tightened using ULT and LPC. Subsequently, the algorithm selects a new constraint from the tightened network, assigns it a label and tests consistency again. This is repeated until either inconsistency

is detected (by ULT or LPC) or a consistent singleton labeling is found. When inconsistency is detected, a dead-end is declared and the algorithm backtracks by undoing the last constraint labeling.

Additional improvements we propose are

(1) to avoid constraint propagation on any subnetwork that is already singly labeled (since it is already consistent);
(2) to avoid using a stack for undoing the last constraint labeling,[4] and instead, to reconstruct the previous partial labeling using of the labels;
(3) to avoid instantiating constraints that were universal in the input network but became non-universal as a result of constraint propagation.

Algorithms ULT and LPC are also useful for preprocessing *before* initiating search. They reduce the number of disjuncts in the constraints, that is the number of intervals in quantitative constraints and the number of allowed relations in qualitative constraints. As a result, the branching factor of the search space is reduced. In addition to reducing the disjunction size, these algorithms render all the universal constraints non-universal. In contrast, using path-consistency algorithms for preprocessing increases the fragmentation and the branching factor.

8. Empirical evaluation

Our empirical evaluation is addressing two questions:

(1) which of the polynomial approximation algorithms presented in this paper is preferable for detecting inconsistencies, and
(2) how effective are these algorithms when used to improve backtrack search via preprocessing, forward checking and dynamic variable ordering.

Section 8.1 presents experiments addressing the first question by measuring the tradeoff between efficiency and effectiveness. Section 8.2 presents experiments addressing the second question.

The problems were generated with the following parameters: n and e are the number of variables and constraints respectively, and k is the number of intervals per quantitative point–point constraint. These quantitative constraints specify integers in the domain $[-R, R]$, and the tightness α of a constraint $T = \{I_1, \ldots, I_k\}$ is $(|I_1| + \cdots + |I_k|)/2R$ where $|I_i|$ is the size of I_i. We used uniform tightness for all constraints. The parameter β is the number of relations in every point–interval constraint and the parameter γ is the number of relations in every interval–interval constraint.

8.1. Comparing constraint propagation algorithms

We evaluate the tradeoff between efficiency and effectiveness of ULT and LPC. Efficiency is measured by comparing the execution time. The effectiveness or accuracy of an algorithm $\mathcal{A}$ is the fraction of times $\mathcal{A}$ returns a correct consistency decision. Since comparing the correct answer by search is too time consuming, we propose to measure

[4] In the stack there would be $O(n^2)$ entries of size $O(n^2)$ each—this was the major problem in [17].

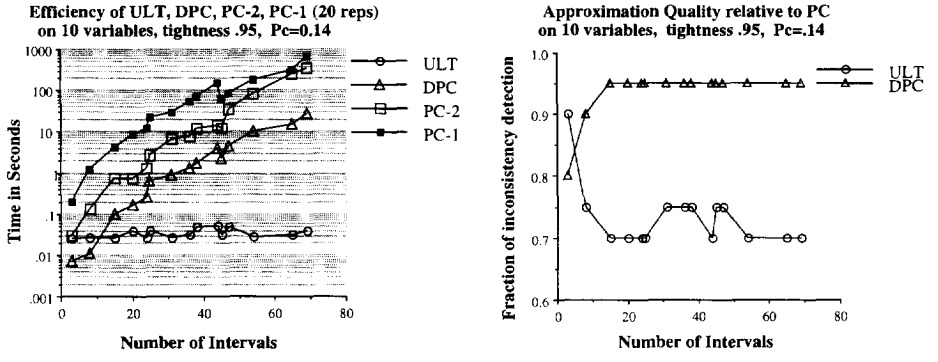


Fig. 18. Execution times and quality of the approximations obtained by DPC and ULT relative to PC. Each point represents 20 runs on networks with 10 variables, 95% tightness.

relative effectiveness instead. To define the notion of relative effectiveness, we rely on the observation that all the approximation algorithms described in this paper are sound, namely when a problem is classified as inconsistent this classification is correct. Thus, two approximation algorithms can differ only in the number of problems they incorrectly classify as consistent. We therefore define the relative effectiveness of two algorithms as the ratio between the number of inconsistencies detected by the algorithms evaluated. In all accuracy plots we use the strong algorithm as the reference point, namely it has 100% accuracy.

8.1.1. Path-consistency versus ULT

In this subsection, we discuss two variants on PC: algorithms PC-1 and PC-2. By PC-1 we refer to the brute-force path-consistency algorithm presented in [8] and by PC-2 we refer to the algorithm presented in Fig. 3(a). We use PC as a collective name for both PC-1 and PC-2.

We next present a quantitative empirical comparison of algorithms PC-1, PC-2, DPC and ULT.[5] In Fig. 18 we show that both PC-1, PC-2 and DPC may be impractical even for small problems with 10 variables. We see that although ULT is orders of magnitude more efficient than PC-1 and PC-2, ULT is able to detect inconsistency in about 70% of the cases that PC-1, PC-2 and DPC detect inconsistencies. Subsequently, we measure the *relative* efficiency–effectiveness tradeoff for ULT and LPC.

8.1.2. Comparing ULT, LPC, DLPC *and* PLPC

Here we measure the relative effectiveness tradeoffs of LPC, ULT, DLPC and PLPC. We test our algorithms on problems having 32 variables. The tightness of interval–interval constraints is 7 relations allowed out of 13, namely the tightness is $\gamma = 7/13$; for point–interval constraints the tightness is $\beta = 4/5$; and for point–point constraints the tightness is $\alpha = 0.45$.

[5] Fig. 14 describes qualitatively the strength of the various algorithms.

Table 1
Effectiveness and efficiency of LPC, DLPC, PLPC and ULT

# of consts	Acc of PLPC	Acc of DLPC	Acc of ULT	# Op. LPC	# Op. PLPC	# Op. DLPC	Time LPC	Time PLPC	Time DLPC	Time ULT
32 vars, 100% interval variables (pure qualitative), 200 reps.										
250	100%	100%	100%	17 K	13 K	11 K	0.621	0.467	0.417	0.621
300	100%	98%	100%	20 K	17 K	15 K	0.748	0.632	0.551	0.748
350	100%	92%	100%	25 K	22 K	19 K	0.886	0.807	0.689	0.886
400	100%	79%	100%	28 K	27 K	23 K	1.001	0.970	0.807	1.001
450	100%	71%	100%	30 K	30 K	26 K	1.056	1.056	0.907	1.056
496	100%	73%	100%	28 K	28 K	25 K	0.971	0.971	0.885	0.971
32 vars, 50% interval variables (mixed), 200 reps.										
150	100%	100%	100%	13 K	6 K	5 K	0.210	0.121	0.082	0.163
200	99%	98%	97%	18 K	11 K	8 K	0.283	0.200	0.135	0.174
250	98%	93%	95%	23 K	17 K	11 K	0.374	0.306	0.199	0.308
300	96%	63%	65%	26 K	22 K	15 K	0.456	0.406	0.266	0.422
350	98%	32%	89%	27 K	25 K	20 K	0.460	0.440	0.325	0.426
400	100%	46%	98%	24 K	23 K	20 K	0.406	0.402	0.347	0.385
450	100%	86%	100%	20 K	20 K	19 K	0.400	0.400	0.343	0.379
496	100%	100%	100%	16 K	16 K	16 K	0.359	0.353	0.294	0.331
32 vars, 100% point variables (pure quantitative), 200 reps.										
150	98%	92%	90%	25 K	12 K	5 K	0.546	0.400	0.165	0.132
200	99%	25%	15%	27 K	17 K	8 K	0.623	0.533	0.259	0.162
250	100%	70%	45%	14 K	11 K	10 K	0.380	0.350	0.315	0.181
300	100%	99%	77%	9 K	8 K	8 K	0.287	0.275	0.270	0.164
350	100%	100%	94%	7 K	7 K	7 K	0.244	0.241	0.235	0.126
400	100%	100%	100%	6 K	6 K	6 K	0.211	0.212	0.204	0.105

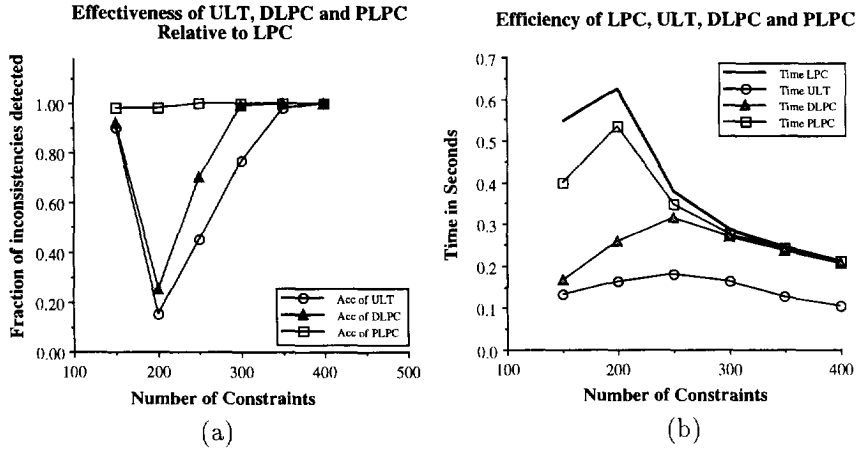


Fig. 19. Effectiveness and efficiency of LPC, ULT, DLPC and PLPC (from Table 1).

The tradeoff between efficiency and effectiveness is presented in Table 1 and is plotted in Fig. 19. Each table entry and data point represents the average of 200 instances. The columns of Table 1 labeled "Acc of $\langle alg \rangle$" specify the accuracy of algorithm $\langle alg \rangle$ relative to LPC, namely the fraction of cases in which algorithm $\langle alg \rangle$ detected inconsistency given that LPC did. The columns labeled "# Op $\langle alg \rangle$" describe the number of revision operations made by algorithm $\langle alg \rangle$. The basic revision operation of PC is $C_{ij} \leftarrow C_{ij} \cap (C_{ik} \odot C_{kj})$. The basic revision operation of LPC is $C_{ij} \leftarrow C_{ij} \lhd (C_{ik} \odot C_{kj})$, while the basic operation for ULT is $C'' \leftarrow C_{ij} \cap ([low(C_{ik}), high(C_{ik})] \odot [low(C_{kj}), high(C_{kj})])$. Measuring the number of revision operations is machine- and implementation-independent, unlike execution time.

For networks with only point variables, having about 200 constraints, ULT detected 15% of the inconsistencies that LPC detected, while DLPC and PLPC detected 25% and 95% inconsistencies respectively. For the same benchmark, the execution time of ULT, DLPC, PLPC, LPC was 0.162, 0.259, 0.533, 0.623 seconds respectively. The general trends in Table 1 indicate that

(1) ULT is clearly the most efficient algorithm, and

(2) PLPC is almost as effective as LPC in detecting inconsistencies.

Based on the results in Table 1 it is difficult to select a clear winner. We speculate that in applications where queries involve a small subset of the variables and efficiency is crucial (e.g. real-time applications, large databases), ULT will be preferable to LPC and its variants. However, on our benchmarks, LPC is by far superior to ULT. Based on experiments made so far, we cautiously conclude that PLPC seems to show the best overall efficiency–effectiveness tradeoff.

8.2. Backtracking

To improve backtrack search, our polynomial approximation algorithms can be used in three ways:

(1) in preprocessing to reduce the number of disjuncts before initiating search,

(2) to perform forward checking (within backtracking) for reduction of fragmentation and early detection of dead-ends, and

(3) as an advice generator to determine the order of constraint labelings.

For simplicity of exposition, we report results of experiments in which the same constraint propagation algorithm is used for preprocessing, forward checking and dynamic variable ordering.

In selecting our benchmark problems, we drew on the recent observation that many classes of NP-complete problems have hard instances in a transition region [4,21]. We therefore identified generation patterns that enable generating problems in the transition region and report the results obtained on those problems. Section 8.2.1 provides results on quantitative TCSPs and Section 8.2.2 provides results on qualitative networks.

8.2.1. Quantitative TCSPs

In general, constraint propagation algorithms are used as a preprocessing phase before backtracking in order to reduce the number of dead-ends encountered during search. When preprocessing with PC, problems become even harder to solve due to increased

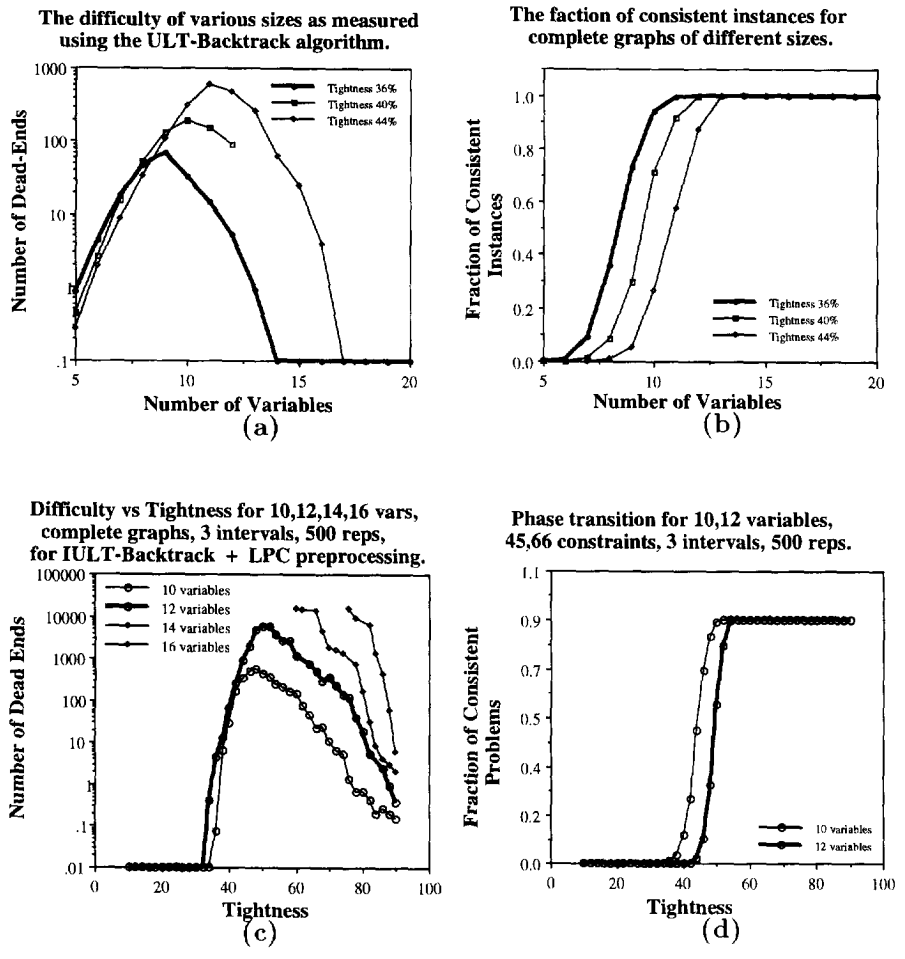


Fig. 20. (a,b) The difficulty as tightness is constant. (c,d) The difficulty as a function of tightness.

fragmentation. In contrast, preprocessing with ULT results in problems on which even naive backtracking is manageable (for small problems). This can be explained from the search space reduction argument mentioned at the end of Section 4.

We compare three backtrack search algorithms: "Old-Backtrack+ULT" which uses ULT as a preprocessing phase with no forward checking and static ordering; "ULT-Backtrack+ULT" and "LPC-Backtrack+LPC" which use ULT and LPC respectively for preprocessing, forward checking and dynamic variable ordering.

The experiments reported in Fig. 20 were conducted with networks of 10–20 variables, complete constraint graphs and three intervals in each constraint. Each point represents 500 runs. The region in which about half of the problems are satisfiable, is called the *transition region* [4,21]. In Figs. 20(a) and 20(b) we observe a phase transition when varying the size of the network, while in Figs. 20(c) and 20(d) we observe a similar phenomenon when varying the tightness of the constraints.

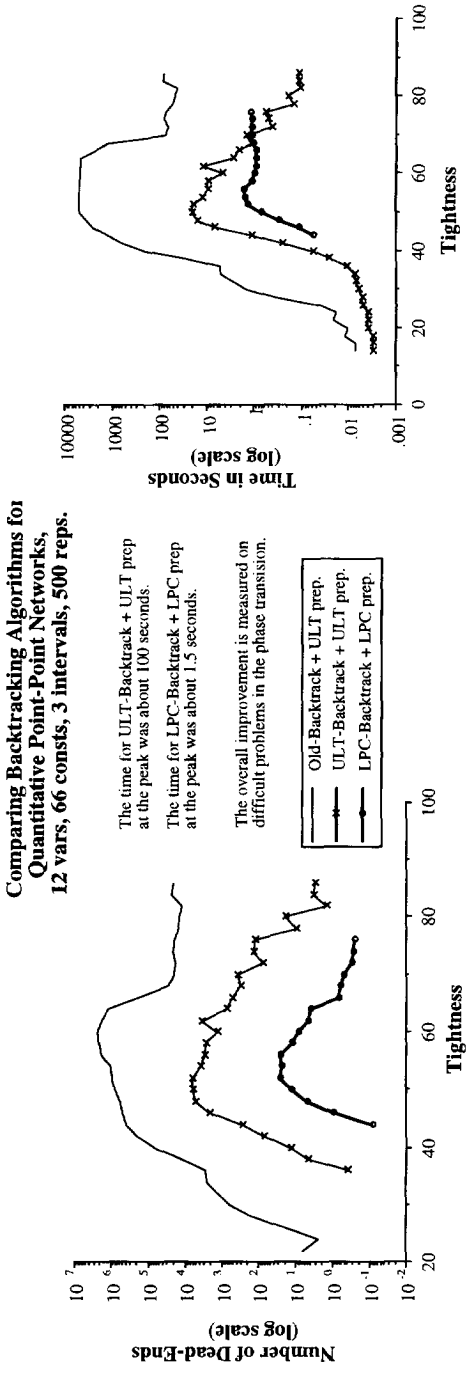


Fig. 21. A comparison of three backtracking algorithms on quantitative TCSPs.

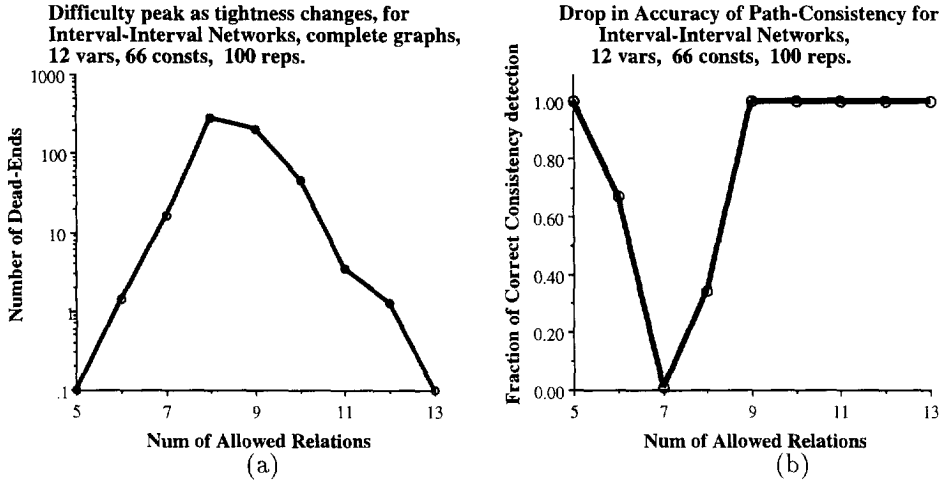


Fig. 22. The difficulty as a function of tightness for qualitative networks.

The experiments reported in Fig. 21 were conducted with networks having 12 variables, 66 constraints (i.e. complete constraint graphs) and three intervals in each constraint. Each point represents 500 runs. ULT and LPC pruned dead-ends and improved search efficiency on our benchmarks by orders of magnitude. Specifically, averaged over 500 instances in the transition region (per point), Old-Backtrack+ULT is about 1000 times slower than ULT-Backtrack+ULT, which is about 1000 times slower than LPC-Backtrack+LPC. The latter encounters about 20 dead-ends on the peak (worst performance). As we depart from the transition region the execution times become smaller and the improvements are less significant.

8.2.2. *Qualitative TCSPs*

Here we present results obtained with backtracking on qualitative TCSPs. We show that

(1) a transition region exists for qualitative networks, and

(2) for problems within this region PC [1] is completely ineffective.

The backtracking algorithm is the algorithm used by Ladkin and Reinefeld [17]. In their implementation, they avoid enforcing path-consistency on any subnetwork that is already labeled (since it is already consistent).

The experiments reported in Fig. 22 were conducted with networks having 12 variables, 66 constraints, and each point is averaged over 100 instances. We change the tightness of the constraints by changing γ. We measure the number of dead-ends (Fig. 22(a)) and the fraction of cases in which enforcing path-consistency correctly decides consistency (Fig. 22(b)).

Fig. 22(a) shows that qualitative networks exhibit a phase transition at $\gamma = 8/13$. The only difference between the experiments reported in this section and those reported in [17] is that the latter used a fixed $\gamma = 0.5$, namely in about half of the cases, six relations out of 13 interval relations were allowed and the other half, seven were allowed.

Our results agree with those reported in [17] in that for $\gamma = 0.5$ most of the generated problems were inconsistent. However, we see that for $\gamma = 9/13$, all the problems generated were consistent. For $\gamma = 6/13$, the problems were about two orders of magnitude easier than those at the peak (Fig. 22(a)) because, in most of the cases, PC detected inconsistency before invoking backtracking search (Fig. 22(b)).

9. Conclusion

Temporal constraint satisfaction problems (TCSPs) provide a formal framework for reasoning about temporal information, which is derived from the framework of classical constraint satisfaction problems (CSPs). As in classical CSPs, the central task of deciding consistency is known to be NP-complete. To cope with intractability it is common to use polynomial approximation algorithms which enforce path-consistency.

In this paper we demonstrated that, in contrast to classical CSPs, enforcing path-consistency on quantitative TCSPs is exponential due to the fragmentation problem. We controlled fragmentation using two new polynomial approximation algorithms, Upper–Lower Tightening (ULT) and Loose Path-Consistency (LPC). When evaluating these algorithms, we addressed two questions empirically:

(1) which of the algorithms presented is preferable for detecting consistency, and
(2) how effective are they when incorporated within backtrack search.

To answer the first question, we measured the tradeoff between efficiency and effectiveness. Efficiency is measured by execution time while effectiveness is measured by counting the fraction of cases in which inconsistency was detected. Using some classes of randomly generated problems, we made two observations:

(1) enforcing path-consistency may indeed be exponential in the number of intervals per constraint, and
(2) ULT's execution time is almost constant in that number.

Nevertheless, ULT is able to detect inconsistency in about 70% of the cases in which path-consistency does. The overall superior algorithm, LPC, is less efficient but more effective than ULT. It is also very effective relative to path-consistency.

To answer the second question, we applied the new algorithms in three ways:

(1) in a preprocessing phase to reduce fragmentation before search,
(2) as a forward checking algorithm for pruning the search, and
(3) as a heuristic for dynamic variable ordering.

We show that for relatively hard problems, which lie in the transition region [4, 21], incorporating ULT within backtracking search is preferred to incorporating path-consistency. Algorithm LPC is superior, in all three roles, as it improves the performance of backtrack search by several orders of magnitude.

Appendix A. ULT for discrete CSPs

The idea of ULT can be extended to approximate path-consistency in classical CSPs. While enforcing full path-consistency requires $O(n^3k^3)$ steps [22], approximating with

Algorithm ULT-CSP

1. **input:** N
2. $N''' \leftarrow N$
3. **repeat**
4. $N \leftarrow N'''$
5. Compute N' by computing the row-convex upper bound of N.
6. Compute N'' by enforcing path consistency on N'.
7. Compute N''' by intersecting N' and N''.
8. **until** $N''' = N$.
9. **if** N''' is consistent, **output:** N"'.
output: "Inconsistent."

Fig. A.1. Algorithm ULT-CSP.

a single iteration of ULT requires $O(n^3k^2)$, and using the complete ULT requires $O(n^3ek + e^2k^2)$. Using a single ULT iteration (weaker than ULT) may significantly reduce propagation time (compared to PC) when the domains are large.

A binary relation R_{ij} on X_i, X_j can be represented by a $(0,1)$-matrix with $|D_i|$ rows and $|D_j|$ columns by imposing an ordering on the domains. A zero entry at row r and column s means that the pair consisting of the rth element of D_i and the sth element of D_j in not allowed.

Definition A.1 (*Row convexity* [31]). A $(0,1)$-matrix is *row convex* iff in each row all of the ones are consecutive, that is no two ones within a single row are separated by a zero in that same row. A constraint is *row convex* iff its matrix representation is *row convex* and the network is *row convex* iff all its constraints are *row convex*. A row convex relation can be represented by a set of k pairs of integers, (l_r, u_r), where r is the row number, l_r is the number of the first non-zero column and u_r is the number of the last non-zero column.

It was shown that enforcing path-consistency on *row convex* networks renders them globally consistent [31]. In Fig. A.1, we present algorithm ULT-CSP. The algorithm relaxes the network into a *row convex* network, enforces path-consistency and intersects the resulting network with the original network, until there is no change.

Definition A.2. Given an arbitrary matrix A, its *upper bound row convex* matrix is obtained by changing, for every row r, all the elements between column l_r and u_r (e.g. $a_{r,l_r} \ldots a_{r,u_r}$) to ones. An upper bound row convex approximation of a binary constraint is obtained by computing an upper bound row convex approximation of its matrix representation. The networks N', N'' and N''' are defined as follows:

- N' is the *row convex upper bound* of N.
- N'' is the minimal network of N' (obtained by enforcing path-consistency).
- N''' is derived from N' and N'' by intersection.

Theorem A.3. *Let N be the input to* ULT-CSP *and R be its output.*

(1) *N and R are equivalent networks.*

(2) *For row convex networks,* ULT-CSP *computes the minimal network in a single iteration.*

(3) *Every iteration of algorithm* ULT-CSP *terminates in* $O(n^3k^2)$ *steps.*

Proof. (1) Let $Sol(N)$ denote the set of solutions of the network N, then $Sol(N) \subseteq Sol(N') = Sol(N'')$. This implies that $Sol(N) \cap Sol(N'') = Sol(N)$ and therefore $Sol(N''') = Sol(N)$.

(2) Clearly, if the input network is row convex, then $N = N'$ and it is known that for row convex networks path-consistency is complete [31].

(3) Computing l_r and u_r for every row in every matrix requires $O(n^2k^2)$ steps and enforcing path-consistency on row convex networks requires $O(n^3k^2)$ steps. □

References

[1] J.F. Allen, Maintaining knowledge about temporal intervals, *Comm. ACM* **26** (1983) 832-843.

[2] J.F. Allen, *Natural Language Understanding* (Benjamin Cummings, Menlo Park, CA, 1987).

[3] M. Boddy, J. Carciofini and B. Schrag, Disjunction for practical temporal reasoning, in: *Proceedings Third International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*, Cambridge, CA (1992).

[4] P. Cheeseman, B. Kanefsky and W. Taylor, Where the really hard problems are, in: *Proceedings IJCAI-91*, Sydney, Australia (1991) 163–169.

[5] P. Dagum, Numeric reasoning with relative orders of magnitude, in: *Proceedings AAAI-93*, Washington, DC (1993) 541–547.

[6] T.L. Dean and D.V. McDermott, Temporal data base management, *Artificial Intelligence* **32** (1987) 1–55.

[7] R. Dechter, From local to global consistency, *Artificial Intelligence* **55** (1992) 87–107.

[8] R. Dechter, I. Meiri and J. Pearl, Temporal constraint satisfaction problems, *Artificial Intelligence* **49** (1991) 61–95.

[9] R. Dechter and J. Pearl, Network-based heuristics for constraint satisfaction problems, *Artificial Intelligence* **34** (1988) 1-38.

[10] A.E. Emerson, A.K. Mox, A.P. Sistla and J. Srinivasan, Quantitative temporal reasoning, in: E.M. Clarke, and R.P. Kurshan, eds., *Computer-Aided Verification*, Lecture Notes in Computer Science **531** (Springer, Berlin, 1990) 136–145.

[11] C.M. Golumbic and R. Shamir, Complexity and algorithms for reasoning about time: graph theoretic approach, Rutcor Research Report 22-91 (1991).

[12] S. Hanks and D.V. McDermott, Default reasoning, nonmonotonic logics, and the frame problem, in: *Proceedings AAAI-86* (1986) 328–333.

[13] K. Kahn and G.A. Gorry, Mechanizing temporal knowledge, *Artificial Intelligence* **9** (1977) 87–108.

[14] H. Kautz and P. Ladkin, Integrating metric and qualitative temporal reasoning, in: *Proceedings AAAI-91* (1991) 241–246,

[15] M. Koubarakis, Foundations of temporal constraint databases, Ph.D. Thesis, National Technical University of Athens (1994).

[16] W. Ho, D.Y.Y. Yun and Y.H. Hu, Planning strategies for switchbox routing, in: *Proceedings International Conference on Computer Design* (1985) 463–467.

[17] P.B. Ladkin and A. Reinefeld, Effective solution of qualitative interval constraint problems, *Artificial Intelligence* **57** (1992) 105–124.

[18] J. Malik and T.O. Binford, Reasoning in time and space, in: *Proceedings IJCAI-83*, Karlsruhe, Germany (1983) 343–345.

[19] D.V. McDermott, A temporal logic for reasoning about processes and plans, *Cognitive Sci.* **6** (1982) 101–155.

[20] I. Meiri, Combining qualitative and quantitative constraints in temporal reasoning, Ph.D. Thesis, UCLA, Los Angeles, CA (1991).

[21] D.S. Mitchell, B. Selman and H.J. Levesque, Hard and easy distributions of SAT problems, in: *Proceedings AAAI-92*, San Jose, CA (1992).

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

[23] B. Nebel and H.J. Burckert, Reasoning about temporal relations: a maximal tractable subclass of Allen's interval algebra, in: *Proc AAAI-94*, Seattle, WA (1994).

[24] M. Poesio and R.J. Brachman, Metric constraints for maintaining appointments: dates and repeated activities, in: *Proceedings AAAI-91*, Anaheim, CA (1991) 253–259.

[25] N. Sateh, Look-ahead techniques for micro-opportunistic job shop scheduling, Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA (1991).

[26] E. Schwalb and R. Dechter, Coping with disjunctions in temporal constraint satisfaction problems, in: *Proceedings AAAI-93*, Washington, DC (1993) 127–132.

[27] E. Schwalb and R. Dechter, Temporal reasoning with constraints on fluents and events, in: *Proceedings AAAI-94*, Seattle, WA (1994).

[28] Y. Shoham, Reasoning about change: time and causation from the stand point of artificial intelligence, Ph.D. Dissertation, Yale University, New Haven, CT (1986).

[29] R.E. Valdéz-Pérez. Spatio-temporal reasoning with inequalities, AIM-875, Artificial Intelligence Laboratory, MIT, Cambridge, MA (1986).

[30] P. van Beek, Reasoning about qualitative temporal information, *Artificial Intelligence* **58** (1992) 297–326.

[31] P. van Beek, Exact and approximate reasoning about qualitative temporal relations, Ph.D. Dissertation, Tech. Rept. TR 90-29, University of Alberta, Edmonton, Alta. (1990).

[32] M. Vilain, H. Kautz and P. van Beek, Constraint propagation algorithms for temporal reasoning: a revised report, in: J. de Kleer and D.S. Weld, eds., *Readings in Qualitative Reasoning about Physical Systems*, 1989.

[33] C.P. Williams and T. Hogg, A typicality of phase transition search, *Computational Intelligence* **9** (1993) 211–238.