Processing Disjunctions in Temporal Constraint Networks *

Eddie Schwalb, **Rina Dechter**
Department of Information and Computer Science
University of California at Irvine, CA 92717
eschwalb,dechter@ics.uci.edu

October 5, 1995

Abstract

The framework of Temporal constraint Satisfaction Problems (TCSP) has been proposed for representing and processing temporal knowledge. Deciding consistency of TCSPs is known to be intractable. As demonstrates in this paper, even local consistency algorithms like path-consistency can be exponential 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. The experiments we performed on hard problems in the transition region show that LPC is the superior algorithm. When incorporated within backtrack search LPC is capable of improving performance by orders of magnitude.

*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

1 Introduction

Problems involving temporal constraints arise in various areas such as temporal databases [6], diagnosis [11], scheduling [22, 21], planning [16], common-sense reasoning [25] and natural language understanding [1]. Several formalisms for expressing and reasoning about temporal constraints have been proposed; interval algebra [2], point algebra [29], Temporal Constraint Satisfaction Problems (TCSP) [7] and the model of combined quantitative and qualitative constraints [17, 12].

The two types of Temporal Constraint Networks that have emerged are qualitative [2] and quantitative [7]. In the qualitative model, variables are time intervals and the constraints are qualitative. In the quantitative model, variables represent time points and the constraints are metric. Subsequently, these two types were combined into a single model [17, 12]. In this paper we build upon the model proposed by [17], whose variables are either points or intervals and involves 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 rational numbers[1], deciding consistency is in NP-complete [7, 17]. For qualitative networks, computing the minimal network is in NP-hard [10, 7]. In both qualitative and quantitative models, the source of complexity stems from allowing disjunctive relationships between pairs of variables. Such constraints often arise in many applications, as demonstrated by the following example:

Example 1 : A large NAVY cargo needs to be transported from New York starting on March 7 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 L.A. the delivery requires 3-4 days by air *or* 13-15 days on the ground. In addition to these constraints, 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) and . The transportation of the AIRFORCE cargo must start between March 17 and March 20 and with duration of 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 these constraints satisfiable?", "can the cargo arrive in L.A. on Jan 8-9?", "when must the cargo arrive in Chicago ?" or "how long may the NAVY cargo transportation take?". The first two queries reduce to deciding consistency and the third query reduces 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. Enforcing

[1] This is always the case in practice.

path-consistency, is the loosest approximation we can provide thus far. As we demonstrate in this paper, in contrast to discrete CSPs, enforcing path-consistency on quantitative TCSPs is exponential. This is because enforcing path-consistency breaks intervals into several smaller subintervals and may result in an exponential blowup, leading to what we call fragmentation.

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

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 in preprocessing, or in guiding the search by forward checking and dynamic ordering.

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 enforcing path-consistency does. Algorithm LPC further improves on ULT and, while being efficient, is capable of detecting almost all 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 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. Using hard problems which lie in the transition region [4, 18], we show that both ULT and LPC are preferred to PC and that LPC is the best algorithm. Using LPC for preprocessing, forward checking and dynamic ordering, improves the performance of backtrack search by several orders of magnitude.

The organization of the paper is as follows. Section 2 presents the model of constraint satisfaction problems and the known algorithms for processing them. Section 3 presents algorithm Upper Lower Tightening (ULT) and section 4 presents Loose Path-Consistency (LPC). Section 5 extends the results presented in sections 3 and 4 to networks of combined qualitative and quantitative constraints. Section 6 presents backtracking algorithms and Section 7 presents an empirical evaluation.

2 Temporal Constraint Networks

2.1 Overview

There are three kinds of Temporal Constraint Satisfaction Problems (TCSP): (1) *qualitative* TCSP, widely known as Allen's Interval Algebra [2], (2) *quantitative* TCSP, introduced in [7], and (3) *combined* qualitative and quantitative TCSP, introduced in [17].

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 variables 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

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

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

1. A point-point constraint between two point variables X_i, X_j is *quantitative*[2] 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 Table 1 for illustration).

3. An interval-interval constraint between two interval variables is *qualitative*, and is in the set
$$\left\{ \begin{array}{c} \textbf{before}, \textbf{after}, \textbf{meets}, \textbf{met-by}, \\ \textbf{overlaps}, \textbf{overlaps-by}, \textbf{during}, \textbf{contains}, \textbf{equals}, \\ \textbf{starts}, \textbf{started-by}, \textbf{finishes}, \textbf{finished-by} \end{array} \right\}$$
abbreviated { b,bi, m,mi, o,oi, d,di, =, s,si, f,fi } respectively (see Table 2 for illustration).

The structure of a TCSP can be represented by a constraint graph. The nodes correspond to variables (point or interval). The arcs connect pairs of directly constrained variables and are labeled by the elements of the disjunctive constraint they represent, namely by sets of intervals (if metric point-point constraints) or by the set of allowed qualitative relations.

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

[2] In Meiri's thesis, a distinction is made between qualitative and quantitative point-point constraints.

Table 1: The 5 qualitative point-interval relations
(X is a point and Y is an interval).

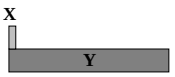
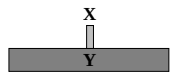
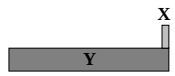
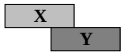
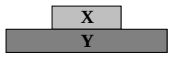
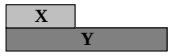
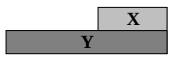
Relation	Symbol	Inverse	Example
X before Y	b	bi	
X starts Y	s	si	
X during Y	d	di	
X finishes Y	f	fi	
X after Y	a	ai	

Table 2: The 13 qualitative interval-interval relations.

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	

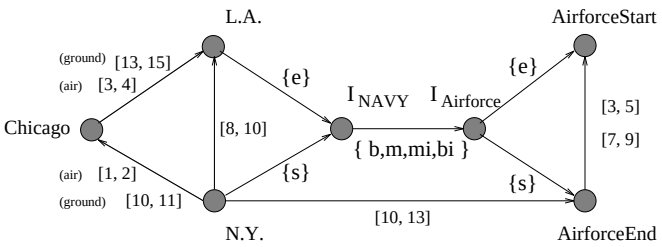


Figure 1: The constraint graph of the logistics problem.

$X_{N.Y.}$ = time point at which the cargo was shipped out of N.Y.,
$X_{Chicago}$ = time point the cargo arrived into and shipped out of CHICAGO
$X_{L.A.}$ = time point at which the cargo arrived into L.A.
I_{NAVY} = transportation period of the NAVY cargo.
$I_{Airforce}$ = transportation period of the AIRFORCE cargo.
$X_{AirforceStart}$ = time point at which the AIRFORCE shipment starts,
$X_{AirforceEnd}$ = time point at which the AIRFORCE shipment ends.

The constraints are:

$$
\begin{array}{l}
X_{N.Y.} - X_0 \in [March7,\ March7] \\
X_{Chicago} - X_{N.Y.} \in [1,2] \cup [10,11] \\
X_{L.A.} - X_{Chicago} \in [3,4] \cup [13,15] \\
X_{L.A.} - X_{N.Y.} \in [8,10] \\
X_{N.Y.}\ \{starts\}\ I_{NAVY} \\
X_{L.A.}\ \{ends\}\ I_{NAVY} \\
X_{AirforceBegin}\ \{starts\}\ I_{Airforce} \\
X_{AirforceEnd}\ \{ends\}\ I_{Airforce} \\
X_{AirforceEnd} - X_{AirforceBegin} \in [3,5] \cup [7,9] \\
X_{AirforceBegin} - X_{N.Y.} \in [10,13] \\
I_{NAVY}\ \{before,\ meets,\ met\text{-}by,\ after\}\ I_{Airforce}
\end{array}
$$

The constraint graph representing this network is given in Figure 1.

2.2 Quantitative TCSPs

For simplicity of exposition, we will present our algorithms for the restricted model of quantitative TCSP first. Thereafter, we extend these algorithm to process Meiri's combined model [17].

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

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

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

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

All intervals are assumed to be open and pairwise disjoint.

Definition 1 : [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.

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

Definition 2 : [minimal network]
A value v is a *feasible value* of X_i if there exists a solution in which $X_i = v$. 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 such that every instantiation of X_i, X_j which is consistent with C_{ij} can be extended to a global solution. A network is minimal iff its domains and constraints are minimal.

A 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 Figure 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).

2.3 Answering Queries

For completeness, we describe the set of queries the 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. What are the possible times event X_i can occur at ?

5. Given the time 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 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 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 is a counter example which shows how can X_i occur less than 5 or more than 10 minutes after X_j.

Queries 4 and 5 can be answered in constant time after computing the equivalent minimal network (defined above) by a simple table lookup. 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 occurred 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.

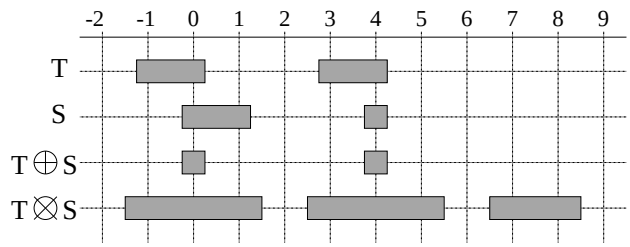
2.4 Constraint Propagation

Since computing the minimal network can be done by a polynomial number of consistency queries, we focus on the task of deciding consistency. Deciding whether a given network is consistent is in NP-complete [7] and deciding whether it is minimal is in NP-hard (which subsumes NP-complete). Therefore, it is common to use algorithms which 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 [8]. For qualitative TCSPs, 3,4-consistency algorithms were covered by [27]. For quantitative TCSPs, 3-consistency, or alternatively path-consistency, is defined using the $\oplus, \otimes$ operations.

Definition 3 : 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 by $T \oplus S$,
   admits only values that are allowed by both of them.

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



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

Figure 2: A pictorial example of the $\oplus$ and $\otimes$ operations.

Note that the $\otimes$ operation may result in intervals that are not pairwise disjoint. Therefore, some additional processing may be required to compute the disjoint interval set.

Definition 4 : The *path-induced* constraint on variables X_i, X_j is $R_{ij}^{path} = \oplus_{\forall k}(C_{ik} \otimes C_{kj})$. A constraint C_{ij} is *path-consistent* iff $C_{ij} \subseteq R_{ij}^{path}$ and a network is *path-consistent* iff all its constraints are *path-consistent*.

Any arbitrary consistent quantitative TCSP can be converted into an equivalent *path-consistent* network by repeatedly applying the relaxation operation $C_{ij} \leftarrow C_{ij} \oplus (C_{ik} \otimes C_{kj})$ until a fixed point is reached. Figure 3 presents an algorithm for enforcing path-consistency. For completeness, we also describe a weaker yet more efficient version of path-consistency, that is tied to a particular ordering of the variables, called DPC [7].

Theorem 1 : *[7]*
If time is described by rational numbers, 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.

In contrast to discrete CSPs, however, enforcing path-consistency on TCSPs is problematic when the range R is large or the domains are continuous [7, 21]. An upper bound on the number of intervals in $T \otimes S$ is $|T| \cdot |S|$, where $|T|, |S|$ 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.

Example 3 : Consider the network presented in Figure 4, having 3 variables, 3 constraints and 3 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 become exponential.

2.5 Simple Temporal Problems

A special class of problems which does not exhibit such an exponential blow-up is 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 $X_j - X_i \leq w_{ij}$, as illustrated in Figure 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 (Figure 5) is equivalent to enforcing path-consistency and is complete for STPs [7]. The focus of the paper is on two algorithms designed to bound the fragmentation.

3 Upper Lower Tightening (ULT)

The intractability of enforcing path-consistency stems from the fact that the relaxation operation $C_{ij} \leftarrow C_{ij} \oplus (C_{ik} \otimes C_{kj})$ may increase the number of intervals in C_{ij}. Our idea is to compute looser constraints which consists of fewer intervals that subsumes all the intervals of the path-induced constraint.

The algorithm for approximating path-consistency, called Upper Lower Tightening (ULT), utilizes the fact that an STP is tractable. The idea is to use the extreme points of all intervals associated with a single constraint as one big interval, yielding an STP, and then to perform path-consistency on that STP. This process can only decrease the number of intervals per constraint. Finally we intersect the resulting simple path-consistent minimal network with the input network.

Definition 5 : (Upper Lower Tightening) Let $C_{ij} = [I_1, \ldots, I_m]$ be the constraint over variables X_i, X_j and let L_{ij}, U_{ij} be the lower and upper bounds of C_{ij}, respectively. We define N', N'', N''' as follows (see Figure 7):

- N' is an STP derived from N by relaxing its constraints to $C'_{ij} = [L_{ij},\ U_{ij}]$.
- N'' is the minimal network of N' (N' is an STP).

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** $C_{ij} \neq C_{ik} \otimes C_{kj}$ **then**
5. $C_{ij} \leftarrow C_{ij} \oplus (C_{ik} \otimes C_{kj})$
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** $C_{ij} \neq C_{ik} \otimes C_{kj}$ **then**
4. $E \leftarrow E \cup (i,j)$
5. $C_{ij} \leftarrow C_{ij} \oplus (C_{ik} \otimes C_{kj})$
6. **if** $C_{ij} = \{\}$ **then** exit (inconsistency)
7. **end-if**
8. **end-for**
9. **end-for**

Figure 3: Algorithms PC and DPC [7].

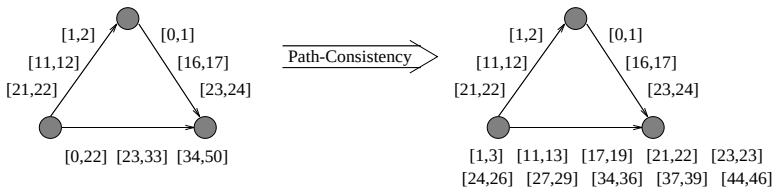


Figure 4: The fragmentation problem.

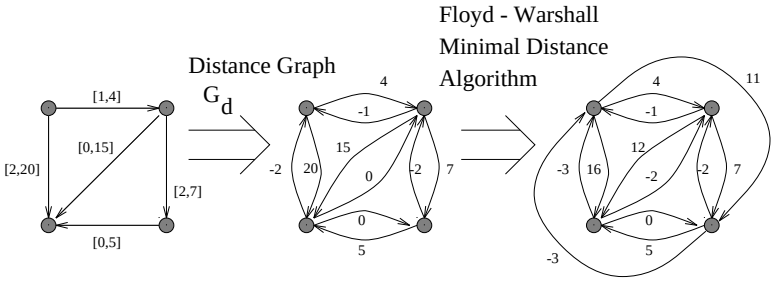


Figure 5: Processing an STP.

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** $\forall ij \quad (L'''_{ij} = L_{ij})$ *and* $(U'''_{ij} = U_{ij})$
   or $\exists ij \quad (U'''_{ij} < L'''_{ij})$
7. **if** $\forall ij \quad (U'''_{ij} > L'''_{ij})$ **output:** N'''
   otherwise **output:** "Inconsistent."

Figure 6: The Upper Lower Tightening (ULT) algorithm.

- N''' is the intersection of N'' and N, namely $C'''_{ij} = C''_{ij} \oplus C_{ij}$.

Algorithm Upper Lower Tightening (ULT) is presented in Figure 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 it with N results in a network which is also equivalent to N.

Lemma 1 : *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: *Part 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 $Sol(N''') = Sol(N)$. Part 2: Let N'_i, N''_i be the networks N', 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 a looser networks than enforcing full path-consistency. A complete comparison, is given in the next section and depicted in Figure 14.

Example 4 : An example run of ULT on a sample problem instance is given in Figure 7. We start with N and compute $N'_{(1)}$, $N''_{(1)}$, $N'''_{(1)}$. Thereafter, we perform the second iteration in which we compute $N'_{(2)}$, $N''_{(2)}$, $N'''_{(2)}$ 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}, allowing inference of new implicit facts that were not specified explicitly in the input network.

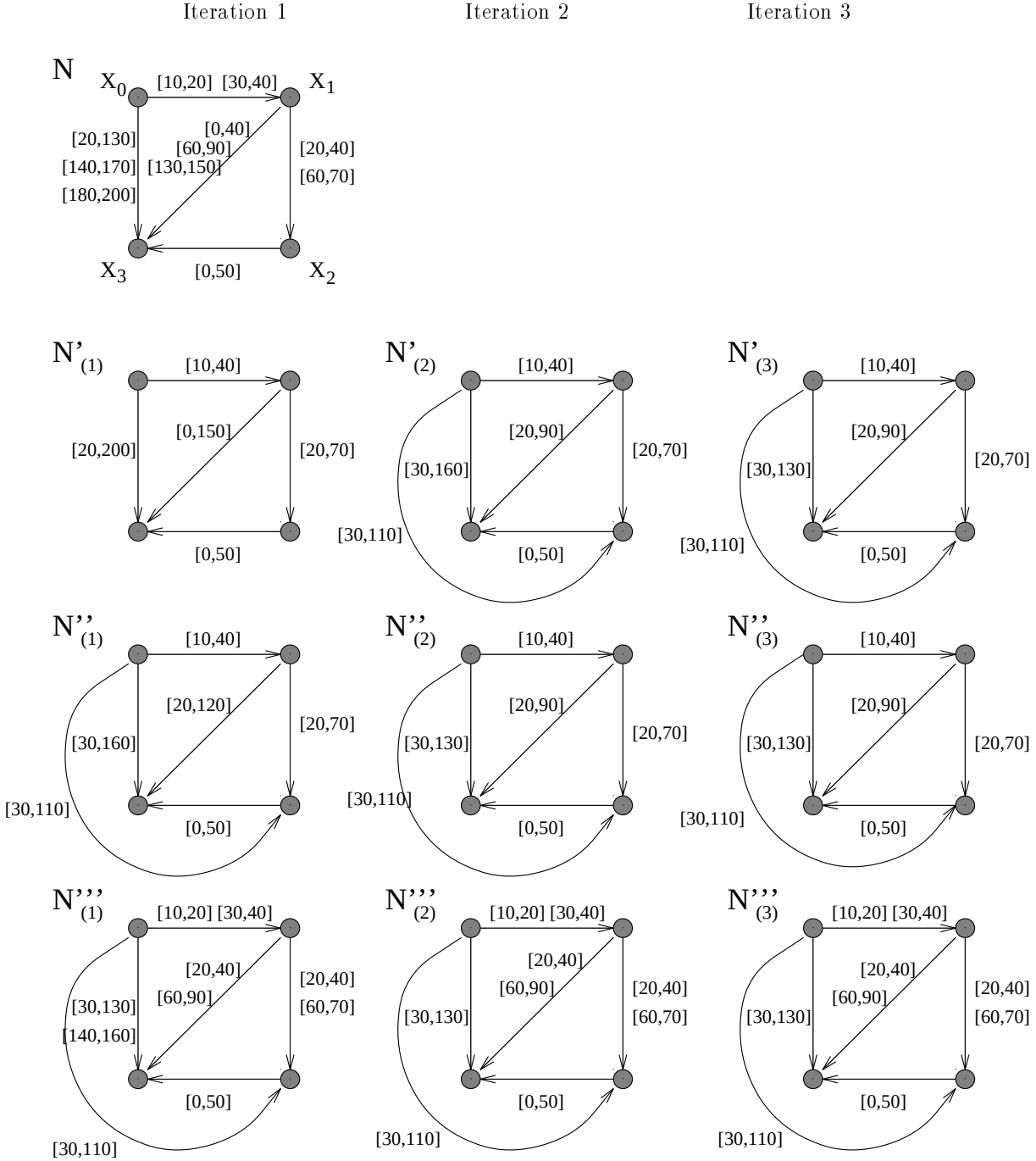


Figure 7: A sample run of ULT.

Theorem 2 : *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: Computing N' requires processing every interval in the network at most once, thus requires $O(ek)$ steps. Computing N'' from N' can be done by applying the all-pairs shortest path algorithm (e.g. Floyd-Warshall) which requires $O(n^3)$ steps. Computing the intersection $T \oplus S$ of two *sorted* constraints requires $O(|T| + |S|)$ steps, thus computing N''' from N'' requires $O(ek)$ steps. As a result, each iteration requires $O(n^3 + ek)$ steps. The halting condition (Figure 6, line 6) implies that at every iteration at least one interval must be removed (Lemma 1). Therefore at most $O(ek)$ iterations are performed yielding a total complexity of $O(n^3ek + e^2k^2)$ steps. □

Note that in contrast to PC, ULT is guaranteed to converge in $O(ek)$ iterations even if the interval boundaries are not rational numbers.

3.1 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 allows parallel execution. We next present two variations on ULT, called ULT-2 and Directional ULT (DULT), which perform such local computations (see Figure 8). We use $low(C_{ij}), high(C_{ij})$ to denote the lowest lower bound and highest upper bound of the union of the intervals in C_{ij}, respectively.

Theorem 3 : *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.*

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

Proof: 1) Algorithm ULT-2 initializes the queue with $O(n^3)$ 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 $T \otimes S$ requires at most $O(k^2)$ steps the total complexity 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 considered in DULT is also considered in ULT-2 but not vice versa and thus DULT is weaker. □

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. $T'_{ij} \leftarrow (C_{ik} \otimes C_{kj})$
5. $T''_{ij} \leftarrow C_{ij} \oplus (low(T'_{ij}), high(T'_{ij}))$
6. **if** $T'_{ij} = \{\}$ **then** exit (inconsistency)
7. **if** $T''_{ij} \neq C_{ij}$ **then**
   $Q \leftarrow Q \cup \{(i,j,k),(k,i,j) \mid 1 \leq k \leq n, i \neq k \neq j\}$
8. $C_{ij} \leftarrow T''_{ij}$
9. **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. $T'_{ij} \leftarrow (C_{ik} \otimes C_{kj})$
4. $T''_{ij} \leftarrow C_{ij} \oplus (low(T'_{ij}), high(T'_{ij}))$
5. **if** $T''_{ij} = \{\}$ **then** exit (inconsistency)
6. **if** $T''_{ij} \neq C_{ij}$ **then** $E \leftarrow E \cup (i,j)$
7. $C_{ij} \leftarrow T''_{ij}$
8. **end-for**
9. **end-for**

Figure 8: Algorithms ULT-2 and DULT.

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 $T'_{ij} = \oplus_{\forall k}(C_{ik} \otimes C_{kj})$, for all i, j.
6. $\quad$ Compute N'' by loosely intersecting $T''_{ij} = C_{ij} \triangleleft T'_{ij}$, for all i, j.
7. **until** $\exists i, j \quad (T''_{ij} = \phi)$; inconsistency, or
   or $\forall i, j \quad |T''_{ij}| = |C_{ij}|$; no interval removed.
8. **if** $\exists i, j \quad (T''_{ij} = \phi)$ **then output** "inconsistent."
   else output: N''.

Figure 10: The Loose Path-Consistency (LPC) algorithm.

4 Loose Path-Consistency (LPC)

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

Definition 6 : Let $T = \{I_1, I_2, \ldots, I_r\}$ and $S = \{J_1, J_2, \ldots, J_s\}$ be two constraints. The *loose intersection*, $T \triangleleft S$ 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 $I_i \oplus S$.

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

Example 5 : Let $T = \{[1, 4], [10, 15]\}$ and $S = \{[3, 11], [14, 19]\}$. Then $T \triangleleft S = \{[3, 4], [10, 15]\}$, $S \triangleleft T = \{[3, 11], [14, 15]\}$ while $S \oplus T = \{[3, 4], [10, 11], [14, 15]\}$.

According to Definition 2, a constraint C_{ij} is path-consistent iff $C_{ij} \subseteq \oplus_{\forall k}(C_{ik} \otimes C_{kj})$. By replacing the intersection operator $\oplus$ with the loose intersection operator $\triangleleft$, we can bound the fragmentation. Algorithm LPC is presented in Figure 10. The network N' is a relaxation of N and therefore loosely intersecting it with N results in an equivalent network.

Example 6 : In Figure 11 we show a trace of LPC on a sample quantitative network. We start with N and compute $N'_{(1)}$, $N''_{(1)}$. Thereafter, we perform a second iteration in which we compute $N'_{(2)}$, $N''_{(2)}$ and finally, in the third iteration, there is no change. The first iteration removes 7 intervals while the second iteration removes a single interval. We see that LPC explicates an induced constraint C_{02}, thus inferring new facts about the time

boundaries that event X_2 can occur. Note that applying ULT on the same network will have no effect and enforcing path-consistency on this sample results in the same network as applying LPC does.

Lemma 2 : *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. □

Theorem 4 : *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, resulting in a complexity of $O(n^3k^3e)$. □

Algorithm LPC computes a tighter networks than ULT. A complete comparison, on an instance by instance basis, is given below and is depicted in Figure 14.

4.1 Variations of LPC

We next present two variations on LPC which have the same structure as PC-2 and DPC. These algorithms, summarized in Figure 13, are called Loose Path-Consistency-2 (LPC-2) and Directional Loose Path-Consistency (DLPC). They differ from PC and DPC only in using the loose intersection $\triangleleft$ operator instead of the strict intersection $\oplus$ operator.

To refine the tradeoff between effectiveness and efficiency, we suggest another variant for constraint propagation. We apply the relaxation operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \otimes C_{kj})$ 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 Figure 14a and the circle network in Figure 14b. The dashed lines point to several triangles which are not processed.

Theorem 5 : *[complexity]*
Given a network N, let n be the number of variables, e be the number of constraints and k be

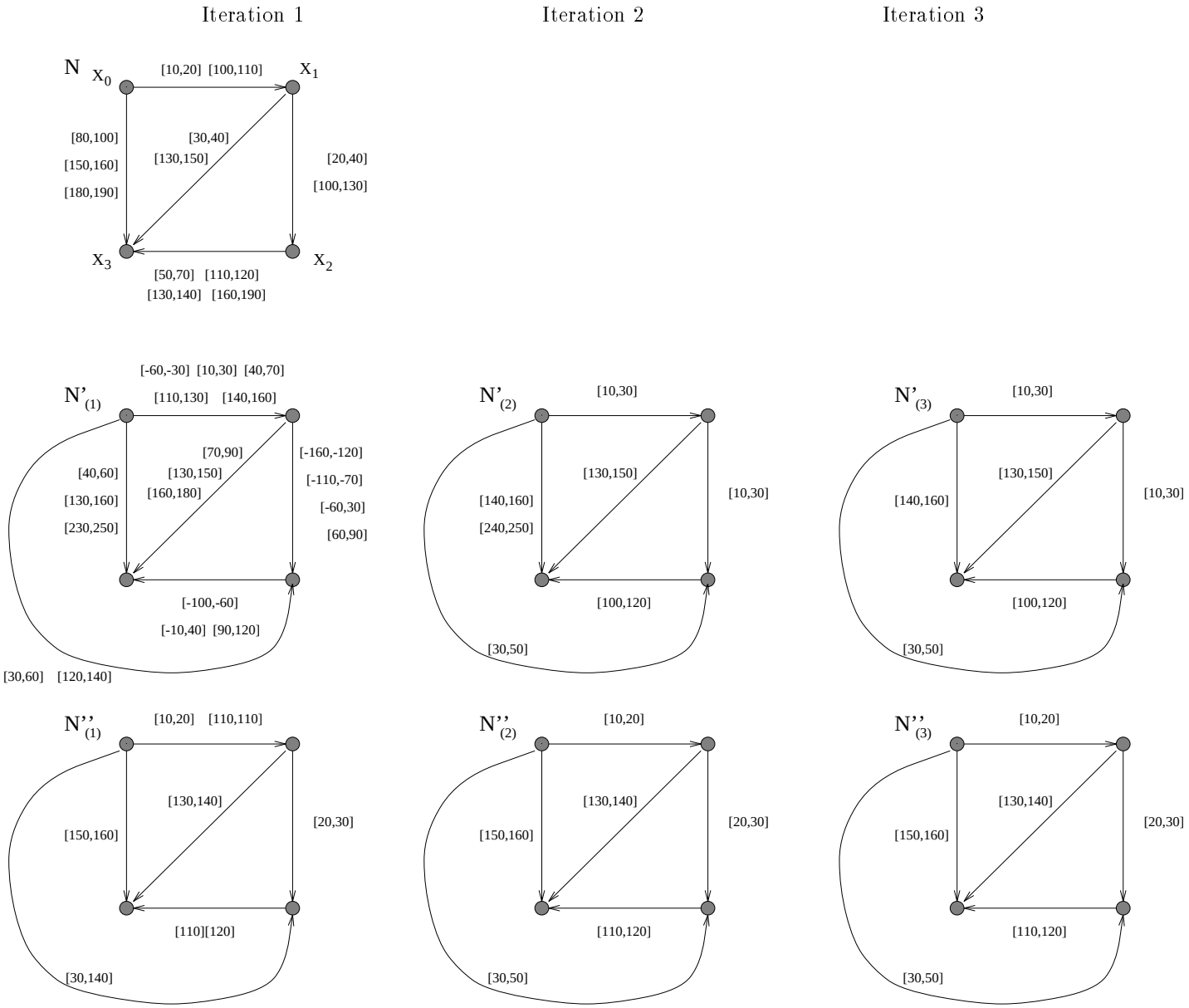


Figure 11: A sample run of LPC.

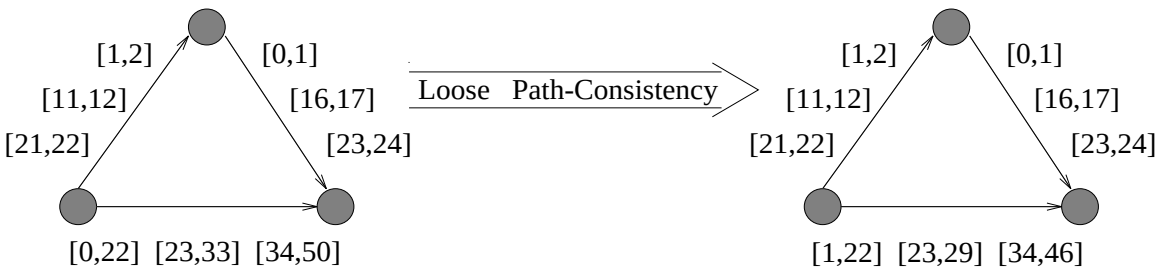


Figure 12: Solving the fragmentation problem.

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. $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \otimes C_{kj})$
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. $T'_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \otimes C_{kj})$
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**

Figure 13: Algorithms LPC-2 and DLPC.

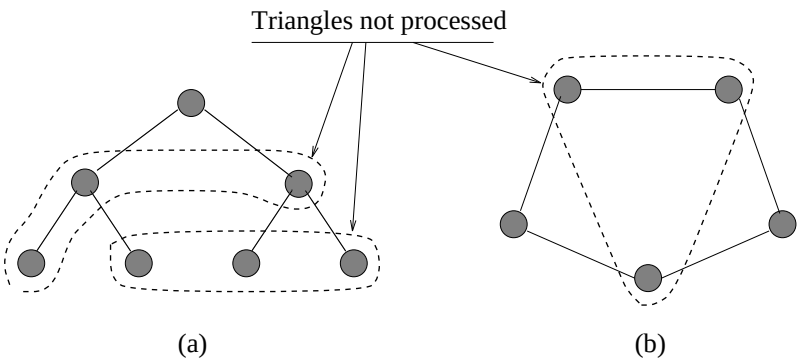


Figure 14: The utility of PLPC.

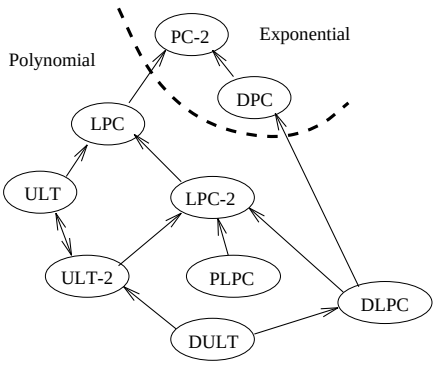


Figure 15: The partial order on the effectiveness.

the maximum number of intervals per constraint. Algorithms LPC-2 and DLPC terminate in $O(nk^2(n^2+ke))$, $O(n^3k^2)$ steps respectively and compute an equivalent network.

Proof: Algorithm LPC-2 applies the operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \otimes C_{kj})$ 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 $T \otimes S$ requires at most $O(k^2)$ steps the total complexity is $O(n^3k^2 + ek^3n)$. Algorithm LPC-2 applies the operation $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \otimes C_{kj})$ at most $O(n^3)$times. Each such operation does not change the set of solutions and requires $O(k^2)$ steps. □

The partial order of all the algorithms presented above is summarized in Figure 15. A directed edge from algorithm $\mathcal{A}_1$ to $\mathcal{A}_2$ indicates that $\mathcal{A}_2$ computes an equal or tighter network than $\mathcal{A}_1$ on an *instance by instance* basis. Note that algorithms PC and DPC are exponential.

5 Extensions to Combined Networks

Algorithms ULT-2, LPC-2 can be extended to process networks of combined qualitative and quantitative constraints. We will describe the extension for LPC only. As defined in Section 2, the combined model involves three types of constraints: point-point (quantitative), point-interval and interval-interval (qualitative). Each node in a triangle can be either a point or an interval variables, resulting in $2^3 = 8$ types of triangles. We therefore modify the semantics of the $\triangleleft, \otimes$ operators to accommodate all 8 combinations types.

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

Case 1: If X_i, X_j, X_k are interval variables then Allen's transitivity table [2] is used to compute $C_{ik} \otimes C_{kj}$ and the $\triangleleft$ operator is interpreted as the usual intersection operator.

Case 2: If both X_i, X_j are interval variables and X_k is a point variable then Meiri's transitivity tables [17] are used to compute $C_{ik} \otimes C_{kj}$ and the $\triangleleft$ operator is interpreted as the usual intersection.

Case 3: If exactly one of X_i, 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 [17] are used to compute $C_{ik} \otimes C_{kj}$; the $\triangleleft$ operator is interpreted as the usual intersection.

Case 4: If X_i, X_j are point variables and X_k is an interval variable then $C_{ik} \otimes C_{kj}$ is computed using the tables given in [17]. If $C_{ik} \otimes C_{kj} \neq \{<, >\}$ then the resulting constraint is translated into a single interval and the $\triangleleft$ operator is interpreted as the $\oplus$ operator in Definition 1. 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 all of X_i, X_j, X_k are point variables then the composition operation used is described by Definition 1 and the $\triangleleft$ operator is described in Definition 3.

With these new definitions of the operators $\otimes, \triangleleft$, we can apply algorithms LPC, LPC-2, DLPC as described in Figures 6 and 9. Algorithm ULT-2 can be extended in a similar manner.

6 General Backtracking.

Algorithms ULT and LPC are useful for detecting inconsistencies and for explicating constraints, however besides being incomplete, they are not designed to find a consistent scenario. 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 constraints and a single relation from a qualitative constraint [17, 7]. Each network 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 [17, 7]. Once inconsistency is detected, the algorithm backtracks. This algorithm can be improved by performing *forward checking* to reduce the number of future possible interval assignment[3] during the labeling process.

Definition 7 : [17] A *basic label* of an arc $i \rightarrow j$ is a selection of (1) a single interval from the interval set C_{ij} for quantitative constraints, and (2) 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 such that *some* constraints are assigned basic labels.

Backtrack search with forward checking is performed in the space of all possible partial labelings as follows: It chooses a disjunctive constraint and replaces it by a single interval or relation 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 algorithm tests consistency using a constraint propagation algorithm like LPC or ULT. Applying ULT or LPC also tightens the network. Subsequently, the algorithm selects a new constraint on the tightened network, assign it a label and test consistency again. This is repeated until either inconsistency is detected or a consistent labeling is found. When

[3] or any other applicable heuristic.

inconsistency is detected, a dead-end is declared and the algorithms backtracks by undoing the last constraint labeling.

Additional improvements we introduce are (1) not to perform constraint propagation on the subnetwork that is already singly labeled (since it is already consistent) (2) not to use a stack for undoing the last constraint labeling[4], and instead, we reconstruct the previous partial labeling using the indexes of the labels; (3) not to instantiate constraints that were universal in the input network but became non-universal as a result of constraint propagation.

In addition to propagating constraints during backtrack search, algorithms ULT and LPC are useful for preprocessing *before* initiating search. These algorithms reduce the number of disjuncts in the constraints, i.e. the number of intervals in quantitative constraints and the number of allowed relation 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 universal constraint non-universal. Note that had we used path-consistency algorithms for preprocessing before search, the fragmentation would have increased. As a result, the branching factor would have been increased and the search would have become less efficient.

7 Empirical Evaluation

Our empirical evaluation is aimed at answering two questions: (1) which of the polynomial approximation algorithms presented above is preferable for detecting inconsistencies, and (2) how effective are these algorithm when used to improve backtrack search via preprocessing, forward checking and dynamic ordering.

Section 7.1 presents experiments aimed at answering the first question by measuring the tradeoff between efficiency and effectiveness. Section 7.2 presents experiments aimed at answering the second question.

Problems were generated with the following parameters: n and e are the number of variables and constraints, and k is the number of intervals per quantitative point-point constraint. These quantitative constraints specify integers in $[-R, R]$, 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 γ is the number of relations in every interval-interval constraint.

[4] In the stack there would be $O(n^2)$ entries of size $O(n^2)$ each - this was the major problem in [14]

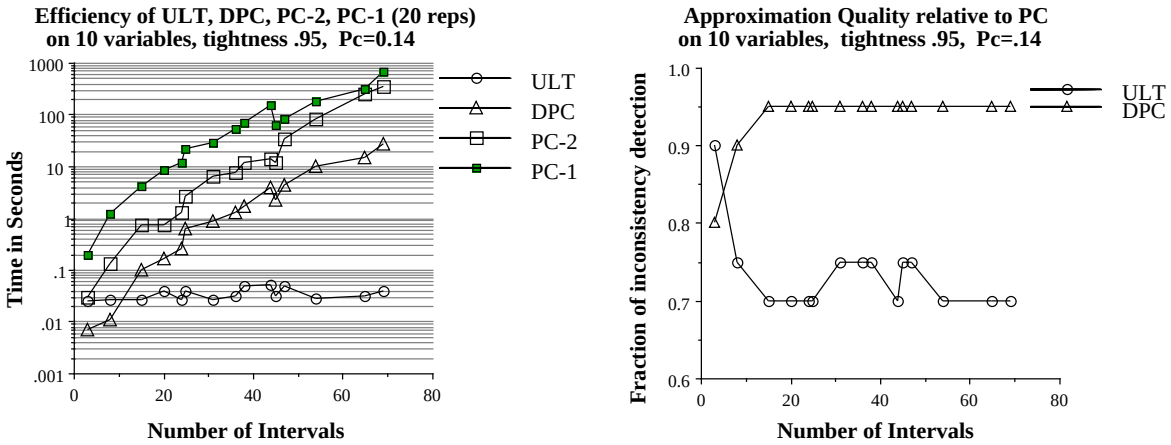


Figure 16: The execution time and quality of the approximation obtained by DPC and ULT to PC. Each point represents 20 runs on networks with 10 variables, 95

7.1 Comparing Constraint Propagation Algorithms

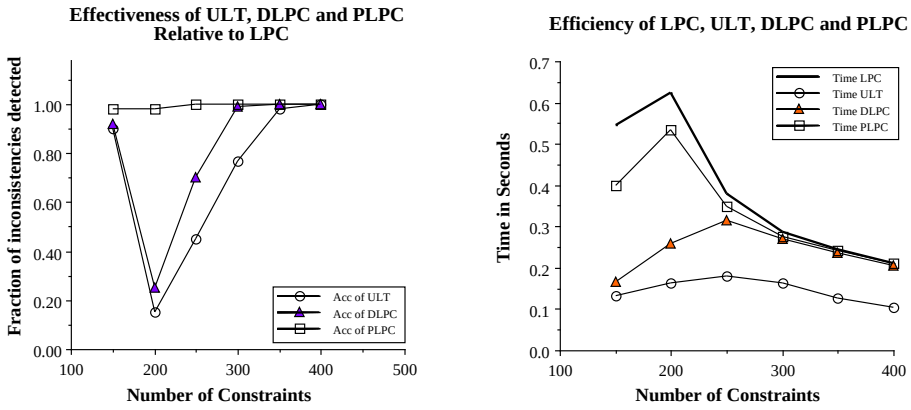
Next we evaluates the tradeoff between efficiency and effectiveness. To compare efficiency, we simply compare execution time. To compare the effectiveness of incomplete constraint propagation algorithms, we propose to compare their effectiveness of by counting the fraction of cases in which the weaker algorithm detected inconsistency given that the stronger one did.

7.1.1 Path-Consistency vs ULT

In Figure 14 we have described the qualitative relationships between the various algorithms. We next present a quantitative empirical comparison of algorithms ULT, LPC and LPC-2. We also include algorithm PC-1 as presented in [7] in this comparison. In Figure 16 we show that despite the fact that ULT is orders of magnitude more efficient than PC, it is able to detect inconsistency in about 70% of the cases that path-consistency does.

7.1.2 Comparing LPC and ULT

The relative effectiveness and efficiency of algorithms LPC, DLPC, PLPC and ULT is presented in Table 1 and Figure 17. The columns labeled "Acc $< alg >$" specify the accuracy of algorithm $< alg >$ relative to LPC, i.e., the fraction of cases algorithm $< alg >$ detected inconsistency given that LPC did. The columns labeled "# Op $< alg >$" describe the number of revision operations made by algorithm $< alg >$. The basic revision operation of LPC is $C_{ij} \leftarrow C_{ij} \triangleleft (C_{ik} \otimes C_{kj})$, while for ULT we use the relaxation operation described



(a) (b)

Figure 17: Effectiveness and Efficiency of LPC, ULT, DLPC and PLPC.

in section 3 definition 3. This measure is machine and implementation independent, unlike execution time.

The problems generated have 32 variables. Tightness of interval-interval constraints is 7 relations allowed out of 13, namely $\gamma = 7/13$; for point-interval constraints $\beta = 4/5$; and for point-point constraints $\alpha = 0.45$. Each entry represents the average of 200 instances.

For networks with only point variables, having about 200 constraints (leftmost column is 200), ULT was capable of detecting 15% of the inconsistencies LPC did (the column labeled "Acc of ULT"), while DLPC and PLPC were capable of detecting 25% and 95% 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 trend we observe in table 1 is that (1) ULT is clearly the most efficient algorithm; (2) PLPC is almost as effective as LPC in detecting inconsistencies.

Based on 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 (real time applications, large databases), ULT will be preferable to LPC and its variants. However, LPC is by far superior to ULT. According to the experience we accumulated so far, we believe that in most cases PLPC is better than LPC.

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

Table 1: Effectiveness and efficiency of LPC, DLPC, Partial LPC and ULT.

7.2 Backtracking

Because the performance of backtrack search is very sensitive to the benchmark being used, generating the correct problem distribution is crucial for obtaining meaningful results. It is commonly hypothesized that for every NP-complete problem, the hard problems lie in a transition region which is similar to the region previously discovered by [4, 18] for SAT problems. We therefore identify that region and use these problems in our experiments.

To improve backtrack search, the new polynomial approximation algorithms can be used in three ways: (1) as a preprocessing phase before initiating search, to reduce the fragmentation, (2) to perform forward checking for early detection of dead-ends, and (3) as an advice generator for dynamic ordering which helps to decide which labeling to perform next. For simplicity of exposition, we report results of experiments in which the same constraint propagation algorithm is used for for preprocessing, forward-checking and dynamic ordering.

In the first part of this section we report results of experiments performed on quantitative TCSPs, while in the second part we focus on qualitative networks.

7.2.1 Quantitative TCSPs

As noted earlier, constraint propagation algorithms can be used as a preprocessing phase before backtracking to reduce the number of dead-ends encountered during search. After preprocessing with algorithm PC, problems become even harder to solve due to the increased fragmentation. In contrast, preprocessing with ULT results in problems on which even naive backtracking is manageable (for small problems).

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 ordering.

The experiments reported in Figure 18 were conducted with networks of 10-16 variables, complete graphs and 3 intervals in each constraint. Each point represents 500 runs. We call the region where about half of the problems are satisfiable, the *transition region* [4, 18]. In Figures 18a and 18b we observe a phase-transition when varying the size of the network, while in Figures 18c and 18d we observe a similar phenomenon when varying the tightness of the constraints.

The experiments reported in Figure 19 were conducted with networks of 12 variables, complete graphs (i.e. 66 constraints) and 3 intervals in each constraint. Each point represents 500 runs. We observe that ULT and LPC are capable of pruning dead-ends and

improving 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) on networks with 12 variables and 66 constraints with 3 intervals to instantiate each. As expected, as we depart from the transition region the improvements are less significant. Note that we could not process these problems using PC due to fragmentation.

7.2.2 Qualitative TCSPs

Next we present results obtained with backtracking on qualitative networks, for which we use the standard path-consistency algorithm [2]. The backtracking algorithm described above is similar to the algorithm used by [14]. In their implementation, they avoid enforcing path-consistency on the subnetwork that is already labeled during backtrack search (since it is already consistent). To this we added forward checking and dynamic ordering, as describe in Section 6.

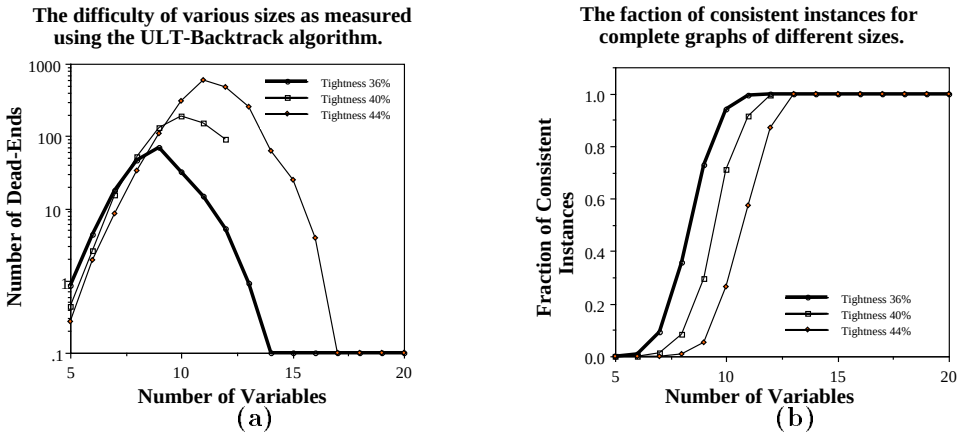
The experiments reported in Figure 20 were conducted with networks of 12 variables, 66 constraints, and each point is averaged over 100 instances. We change the tightness of the constraints by changing γ. The parameters we measure are the number of dead-ends (Figure 20a) and the fraction of cases enforcing path-consistency correctly decides consistency (Figure 20b).

In Figure 20a we show that qualitative networks exhibit a phase transition at $\gamma = 8/13$. The only difference between the experiments reported in this section and those conducted by [14] is that the latter used a fixed $\gamma = 0.5$, namely in about half of the cases, six interval relations out of 13 were allowed and in another half, seven were allowed.

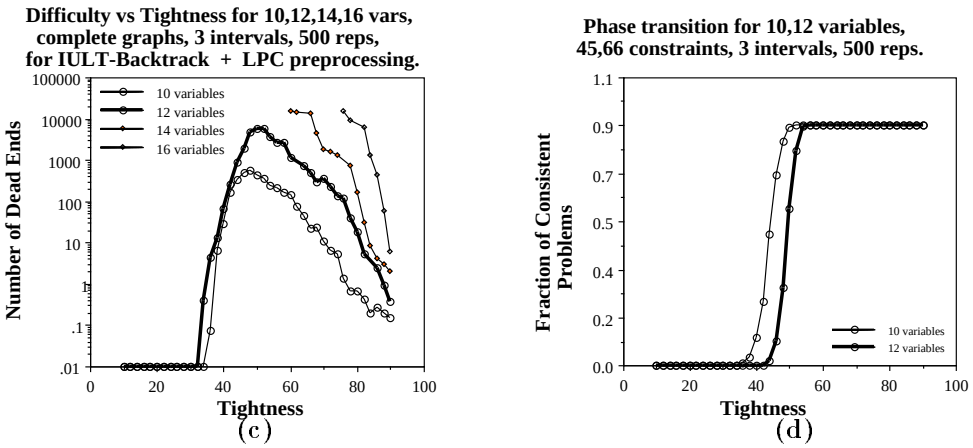
Our results agree with those reported in [14] in that for $\gamma = 0.5$ most of the generated problems were inconsistent. However, we see that for $\gamma = 9/13$, all the problems generated were consistent. For $\gamma = 6/13$, the problems were about two orders of magnitude easier than those at the peak (Figure 20a) because, in most of the cases, path-consistency detects inconsistency before invoking backtracking search (Figure 20b).

8 Conclusion

Temporal Constraint Satisfaction Problems (TCSP) provide a formal framework for reasoning about temporal information, which is derived from the framework of classical 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.



The difficulty as tightness is constant.



The difficulty as a function of tightness.

Figure 18

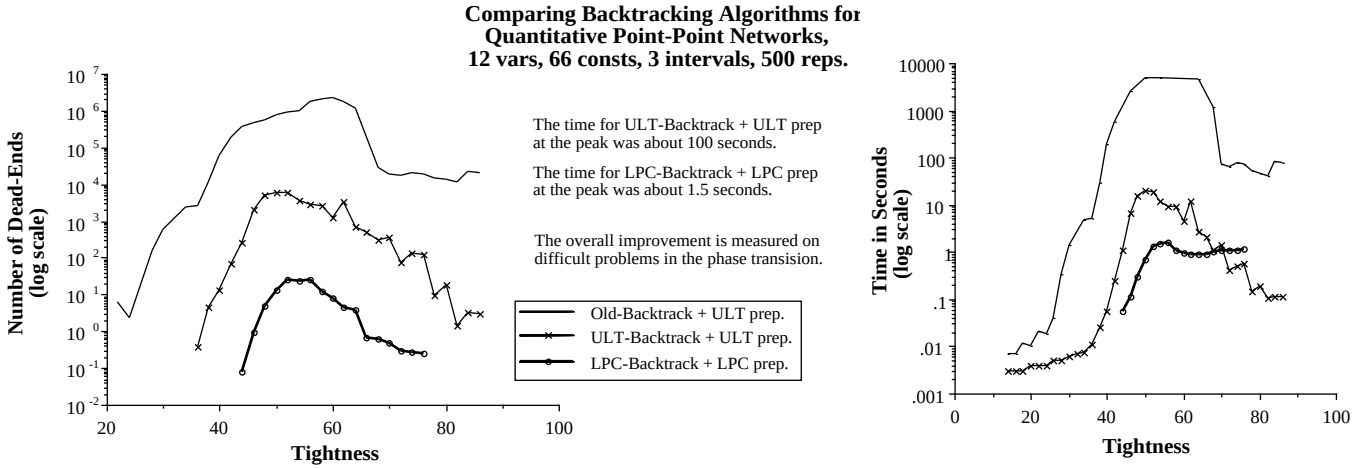


Figure 19: A comparison of various backtracking algorithms.

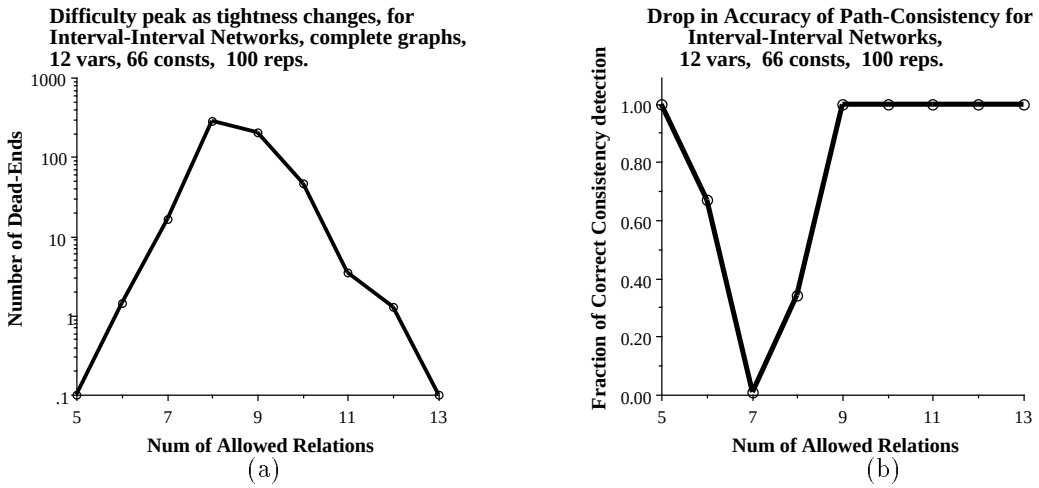


Figure 20: The difficulty as a function of tightness for Qualitative Networks.

In this paper we demonstrated that, in contrast to classical CSPs, enforcing path-consistency on quantitative TCSPs is exponential due to the fragmentation problem. To control fragmentation, we present two new polynomial approximation algorithms, Upper lower Tightening (ULT) and Loose Path-Consistency (LPC), which are effective in detecting inconsistencies and reducing the fragmentation. We present refinements on the main algorithms called ULT-2, Directional ULT (DULT), LPC-2, Directional LPC (DLPC) and Partial LPC (PLPC).

We addressed two questions empirically: (1) which of the algorithms presented is preferable for detecting consistency, and (2) how effective are these algorithms when incorporated within backtrack search.

To answer the first question, we are measure 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.

We show that on randomly generated problems, enforcing path-consistency may indeed be exponential in the number of intervals per constraint while 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 enforcing path-consistency does. The best algorithm, LPC, is less efficient than ULT, is more effective than ULT and very effective relative to enforcing path-consistency.

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) as 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. Using hard problems which lie in the transition region [4, 18], we show that incorporating ULT in backtrack search is preferred to incorporating PC and that LPC is the best algorithm, in all three applications. Using LPC for preprocessing, forward checking and dynamic ordering, improves the performance of backtrack search by several orders of magnitude.

References

[1] Allen, J.F., 1987. Natural Language Understanding, Benjamin Cummings.

[2] Allen, J.F., 1983. Maintaining knowledge about temporal intervals, *CACM* 26 (11): 832-843.

[3] Boddy, M., Carciofini, J., Schrag, B., 1992. Managing Disjunction for Practical Temporal Reasoning, In Proc. *KR-92*.

[4] Cheesman, P, Kanefsky, B., Taylor, W., 1991. Where the Really Hard Problems Are. Proc. of IJCAI-91, 163-169.

[5] Dague, P., 1993. Numeric Reasoning with Relative Orders of Magnitude, Proc. of AAAI-93, 541-547.

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

[7] Dechter, R., Meiri, I., Pearl, J., 1991. Temporal Constraint Satisfaction Problems, *Artificial Intelligence* 49(1991) 61-95.

[8] Dechter, R., "From Local to Global Consistency" *Artificial Intelligence*, 55 (1992), 87-107.

[9] Emerson, A.E., Mox, A.K., Sistla, A.P., Srininasan, J., 1990. Quantitative Temporal Reasoning, Lecture Notes in Computer Science 531, Computer-Aided Verification, Clarke, E.M., Kurshan, R.P. (Eds), 136-145.

[10] Golumbic, C.M., Shamir, R., 1991. Complexity and Algorithms for Reasoning about Time: AYou wanted to meet today. Let me know when? graph theoretic approach, Rutcor Research Report 22-91 (May 1991).

[Hanks & McDermott 86] Hanks, S., McDermott, D.V., 1986. Default reasoning, nonmonotonic logics, and the frame problem. In Proc. of AAAI-86, 328-333.

[11] Kahn,K., Gorry, G.A., Mechanizing Temporal Knowledge, *Artificial Intelligence* **9**(1977)87-108.

[12] Kautz, H., Ladkin, P., 1991. Integrating Metric and Qualitative Temporal Reasoning, In *Proc. of AAAI-91*, pages 241-246, 1991.

[13] Ho, W., Yun, D.Y.Y., Hu, Y.H., 1985. Planning Strategies for Switchbox Routing, In Proc. of Intl. Conf. on Computer Design 1985, 463-467.

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

[15] Malik, J., Binford, T.O., 1983. Reasoning in time and space, In Proc. of IJCAI-83, 343-345.

[16] McDermott, D.V., 1982. A Temporal Logic for Reasoning about Processes and Plans, *Cognitive Science* 6(1982) 101-155.

[17] Meiri, I., 1991. Combining Qualitative and Quantitative constraints in temporal reasoning Ph.D. Thesis, UCLA 1991.

[18] Mitchell, D., Selman, B., Levesque, H., 1992. Hard and Easy Distributions of SAT Problems, Proc. of AAAI-92.

[19] Mohr, R, Henderson, T.R., 1986. Arc and path consistency revisited, *Artificial Intelligence* **28** (2) (1986) 225-233.

[20] Nebel, B., Burckert, H.J., 1994. Reasoning about Temporal Relations: A Maximal Tractable Subclass of Allen's Interval Algebra, In Proc of AAAI-94.

[21] Poesio, M., Brachman, R. J. 1991. Metric Constraints for Maintaining Appointments: Dates and Repeated Activities. In Proc. AAAI-91, 253-259.

[22] Sateh, N., 1991. Look-Ahead techniques for Micro-opportunistic Job Shop Scheduling, Ph.D. thessis, Schoold of Computer Science, Carnegie Mellon University, March 1991.

[23] Schwalb, E., Dechter, R., 1993. Coping with Disjunctions in Temporal Constraint Satisfaction Problems, In Proc. AAAI-93, 127-132.

[24] Schwalb, E., Dechter, R., 1994. Temporal Reasoning with Constraints on Fluents and Events, In Proc. AAAI-94.

[25] Shoham, Y., 1986. Reasoning about Change: time and causation from the stand point of artificial intelligence, Ph.D. dissertation, Yale Univ. 1986.

[26] Valdez-Perez. R.E., 1986. Spacio-Temporal Reasoning with inequalities, Artificial Intelligence Laboratory, AIM-875, MIT, Cambridge (1986).

[27] Van Beek, P., 1992. Reasoning about Qualitative Temporal Information, *Artificial Intelligence* 58 (1992) 297-326.

[28] Van Beek, P., 1990. Exact and Approximate Reasoning about Qualitative Temporal Relations, Ph.D. Dissertation, Tech-Rep TR 90-29, University of Alberta.

[29] Vilain, M., Kautz, H., Van Beek, P., 1989. Constraint Propagation Algorithms for Temporal Reasoning: A revised Report. In Readings in Qualitative Reasoning about Physical Systems, J. de Kleer and D. Weld (eds). 1989.

[30] Williams, C.P., Hogg, T., 1993. A typicality of phase transition search, *Computational Intelligence* 9(3):211-238.

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 [19], approximating with 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 r-th element of D_i and the s-th element of D_j in not allowed.

Definition 8 : (row convexity [28]) 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 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 [28]. In Figure 9, 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 9 : 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 of its matrix representation. The networks N', N'', 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 6 : *Let N be the input to ULT-CSP and R be its output.*

1. *N and R are equivalent networks.*

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."

Figure 9: Algorithm ULT-CSP.

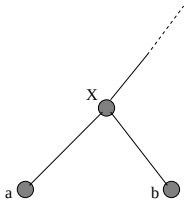


Figure 11: Sample portion of the search tree

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: Part 1: Let $Sol(N)$ denote the set of solutions of then $Sol(N) \subseteq Sol(N') = Sol(N'')$. This implies that $Sol(N) \cap Sol(N'') = Sol(N)$ and therefore $Sol(N''') = Sol(N)$. Part 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 [28]. Part 3: Computing l_r, 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. □

B Detailed Backtracking Algorithms

The key for scaling up the backtrack algorithm suggested above is effective memory management. In contrast to classical CSPs, when backtracking on TCSPs there is a need to store information about the complete network at each level of the search tree.

To illustrate the problem, consider part of the search tree shown in Figure 11, in which every

node is a *partial labeling*. Suppose the search algorithm expands node "X", and thereafter visits the left child labeled by "a". Suppose that once node "a" was visited there is a need to backtrack due to inconsistency. The naive way to allow backtracking is to simply store the partial labeling of "X". Such an approach, however, requires to store all partial labelings on the path from the root of the search tree to the current node, which may require $O(n^4)$ space[5]. Instead, we propose to construct the partial labeling of node "b" during search without storing or reconstructing "X". We store only the necessary information required for reconstruction of node "b", namely the index of the basic labels within every constraint. Note that applying LPC removes some intervals from the constraints and therefore such an indexing should be carefully handled. The saving obtained by this method is mostly apparent when the common parent of a and b is several levels up, closer to the root of the tree.

When a dead-end is encountered, we determine the source of the conflics as follows. Suppose the dead-ends occurred at the constraint C_{ij}, namely, instantiating C_{ij} with any of its intervals renders the networks inconsistent. Suppose the constraint instantiated before C_{ij} was C_{pq}. Then if the networks in which C_{pq} is made universal is inconsistent with every possible instantiation of C_{ij} then C_{pq} is clearly not responsible for the dead-end. In that case, we check the constraints instantiated before C_{pq} by making it universal and enforcing path-consistency, until we find a constraint for which path-consistency does not detect inconsistency.

The complete backtracking algorithm is presented in Figure 16. The function of $LabelNetwork$, shown in Figure 16, is to reconstruct the partial labeling based on the indexes. It receives as input the original network N (the root of the tree), the indexes of the basic labels to be selected from each constraint stored in the $Index$ matrix, and the last constraint which to be instantiated, C_{ij}. Two copies of the network are maintained: N is the original input network and N' is the partial labeling currently expanded; the ij-th constraint of N' is denoted by T'_{ij}.

In contrast to Ladkin and Reinefeld, we propose to perform limited propagation. As shown in Figure 15 lines 5-8, because every iteration of the "repeat" loop removes at least one atomic relation from T'_{ij} (otherwise no change) we perform at most $max(26n, 2nk)$ relaxation operations where n is the number of variables and k is the maximal number of intervals in a point-point constraint. In average, however, we perform much less than $2nk$.

Finally, the last improvement we propose is not to instantiate constraints that were initially universal. It is easy to see that every consistent labeling of all the non-universal constraints is also consistent with the universal constraints; as a result, unnecessary dead-ends can be avoided.

[5] an entry for every constraint - $O(n^2)$ entries; each entry describes a complete network - $O(n^2)$ space each.

Input: A network N with n variables.
Output: A consistent *singleton labeling* of N if consistent, or
a notification that N is inconsistent.

1. $i \leftarrow 0$; $j \leftarrow 1$
2. **repeat**
3. $N' \leftarrow LabelNetwork(N, Index, i, j)$
4. $Index[i,j] \leftarrow 0$
5. **repeat**
6. $\forall k \in [j+1, n]$ $T'_{ik} \leftarrow T'_{ik} \triangleleft (T'_{ij} \otimes T'_{jk})$
7. $\forall k \in [j+1, n]$ $T'_{kj} \leftarrow T'_{kj} \triangleleft (T'_{ki} \otimes T'_{ij})$
8. **until** no change.
9. $Length[i,j] = |T'_{i,j}|$
10. **if** N' is inconsistent **then** let i,j be the *previous* i,j such that $i < j$.
11. **while** $Index[i,j] \geq Length[i,j]$ and $j > 0$ **do**
12. let i,j be the *previous* i,j such that $i < j$.
13. **if** $j > 0$ **then**
14. $Index[i,j] = Index[i,j] + 1$
15. let i,j be the next constraint to be instantiated such that $i < j$.
16. **until** N' is a consistent *singleton labeling* or $j = 0$.
17. **if** N' is a consistent *singleton labeling* **then** exit with N' as the solution.
18. **else** exit with failure.

Figure 15: The general Backtracking algorithm.

LabelNetwork(N, $Index$, i,j**)**
Input: N, a network with n variables,
$Index$, the indexes of the labels,
i,j, the current constraint not to be instantiated.
Output: N', a *partial labeling* of N.

1. $\forall q \in [0, j-1]$, $\forall p \in [0, q-1]$, $T'_{pq} \leftarrow$ *the* $Index[p,q]$*-th label of* C_{pq}.
2. $\forall q \in [0, i-1]$, $T'_{jq} \leftarrow$ *the* $Index[j,q]$*-th label of* C_{jq}.
3. return(N').

Figure 16: Reconstructing a partial labeling.