Fast Algebraic Methods for Interval Constraint Problems*

Peter B. Ladkin
Universität Bielefeld
Technische Fakultät
Postfach 10 01 31
D-33501 Bielefeld, Germany.
ladkin@techfak.uni-bielefeld.de

Alexander Reinefeld
Paderborn Center for Parallel Computing
Fürstenallee 7
D-33095 Paderborn, Germany
ar@uni-paderborn.de

21 August 1996

An invited paper to appear in the *Annals of Mathematics and Artificial Intelligence*

Abstract

We describe an effective generic method for solving constraint problems, based on Tarski's relation algebra, using path-consistency as a pruning technique. We investigate the performance of this method on interval constraint problems. Time performance is affected strongly by the path-consistency calculations, which involve the calculation of compositions of relations. We investigate various methods of tuning composition calculations, and also path-consistency computations. Space performance is affected by the branching factor during search. Reducing this branching factor depends on the existence of 'nice' subclasses of the constraint domain. Finally, we survey the statistics of consistency properties of interval constraint problems. Problems of up to 500 variables may be solved in expected cubic time. Evidence is presented that the 'phase transition' occurs in the range $6 \leq n.c \leq 15$, where n is the number of variables, and c is the ratio of non-trivial constraints to possible constraints.

1 Introduction

Constraint satisfaction problems, or CSPs [Mac87] are a form of logic programming (see Section 2), and are usually solved by various tuned search methods through tuples of values (e.g. [Güs89]). Interval constraints form a particular class of constraint problems that *prima facie* may not be solved using these methods, since the domains of the relations are infinite. Interval constraint problems are found in planning and scheduling in AI, and solving such problems is essential for the mixed qualitative-quantitative problems studied in [KL91]. Allen originally showed how to use search-pruning techniques based on path-consistency, performed as symbolic computations with the constraint relations themselves rather than on tuples of values [All83], and later research has built on this insight (e.g. [Lad90] contains a survey of work to 1989, and see [vB92] for more recent work).

Ladkin and Maddux showed that the relation algebra of Tarski is an appropriate mathematical context in which to represent and solve binary constraint problems [LM88, LM94]. An *algebra of relations* is a set of binary relations closed under the Boolean operations (i.e. forming a Boolean algebra), with the additional operations of *composition* and *converse*, and

*This paper is an extended version of [LR93], including results to date.

containing an *identity* relation (an algebraic identity element for the composition operation). All these operations are used in constraint satisfaction techniques for binary constraints. Sections 2 and 3 are an introduction to this symbolic approach to constraint satisfaction problems. Every finite algebra of relations contains a collection of *minimal non-empty* relations, called *atoms* or *atomic relations*, such that every relation in the algebra may be expressed as a sum of atoms.

We present a general algorithm architecture in which constraint problems over various domains may be solved symbolically (Figure 12). Our design relies on highly-pruned search, and is suitable for NP-hard constraint problems in which search is a component of a solution algorithm. Path-consistency preprocessing is used, followed by a search which involves path-consistency pruning at each choice point. Our design may be used when there is a class of constraint relations C such that

(a) the composition table for members of C may be stored and accessed efficiently;

(b) there is an efficient[1] method for solving problems whose constraints are all in C;

(c) all constraint relations in the problem domain may be represented as unions of members of C.

The last condition (c) entails that the atomic relations are members of C, and so if conditions (a) and (b) hold for the atomic relations, they form the minimal class C for which (a) - (c) hold. Our algorithm takes as input a constraint problem in the domain, and returns a more restricted problem in which all constraints are relations in C that are potentially tighter than the originals. One may then apply the C-constraint-solver (condition (b)) to complete the solution. Although C = *atomic relations* will work if any class C does, it is preferable to choose class C as large as possible satisfying (a) - (c), to reduce the branching factor in search and to reduce the time taken in calculations of compositions (see below).

Interval constraint problems fit this paradigm, as do various spatial reasoning methods (see **Recent Related Work** below). Solving interval constraint problems is NP-complete [VKvB89, Mad94], so some search must be expected in a solution algorithm (unless P=NP). There are 13 atoms (atomic relations) in the Interval Algebra, giving a composition table of 169 entries (144 entries are actually used since one of the atoms is an identity for composition); and problems with atomic constraints may be solved deterministically in quadratic time. There is also a larger class C with 187 members, namely the *pointisable relations*, fulfilling conditions (a)–(c) (see Section 2 for a definition of *pointisable*). Hence we can compare performance of the algorithm design with C = *atoms* with that of C = *pointisables*. The pointisable relations were introduced independently in [LM88] and in [vBC89]. A quadratic time algorithm for solving interval problems with either atomic constraints or pointisable constraints may be derived from [vB90]. Nebel and Bürckert discovered the maximal subclass fulfilling conditions (a)–(c), called the ORD-Horn relations [NB95]. There are 868 ORD-Horn relations, and ORD-Horn constraint problems may be solved in asymptotic cubic time.

Our generic algorithm design works with a finite algebra of constraint relations. There are important infinite classes of constraints, for example numerical constraints such as $a \leq x - y \leq b$ (in which a, b are real numbers) of [DMP91]. This class does not satisfy condition (a) above, but nevertheless there is an efficient way to calculate the composition of two constraints, and in fact solutions may be obtained in asymptotic cubic time.

In previous experiments on interval problems, we found that path-consistency is particularly effective as a preprocessing method, especially on large problems [RL92], and that when

[1] By 'efficient' here we mean low polynomial time, e.g. quadratic time.

used also for incrementally pruning the search, reduces the search almost to a linear selection in most cases [LR92]. Our results were reported for *completion coefficient* approximately equal to 1, and $C = atoms$. The *completion coefficient* is the ratio of actual non-trivial constraints in a problem to the number of possible constraints. For random problems, algorithm versions return atomic reductions extremely fast. 1000 problems may be solved in 3 seconds or so on a notebook 486-based computer running Linux and *gcc*. The fast solution time seems to be due to (a) the pruning power of path-consistency; (b) the now generally-recognised 'phase transition' feature of NP-complete problems [Sel95]: in this case, with extremely high probability, randomly generated interval networks with $n.c \geq 14$ are inconsistent, which is rapidly detected by a path-consistency computation, whereas small networks are consistent with dense solution nodes, entailing that there is little backtracking on either small or large problems.

We show here (as in the 'conference version' of this paper [LR93]) that the conclusions of [LR92] are also valid for varying values of the completion coefficient $c \geq 0.5$, namely that problems with non-trivial search lie generally in the range $6 \leq n.c \leq 15$ for this range of c. To obtain the conclusions for $c = 1$, we had run large-scale experiments on over $200,000$ networks of sizes $3 \leq n \leq 20$. Constraint problems were generated using a distribution in which, for every individual constraint in a problem, each possible relation had equal probability of being that constraint. We performed similar experiments for other values of c. These results are summarised in Figure 16 with the curve $c = 1$. We also performed experiments for $C =$ *pointisable relations*. This reduced the branching factor in search, and therefore enabled us to study behavior on large problems (see below).

By comparing times taken on smaller problems with $C = atoms$ with that taken on the same problems with $C = pointisables$, we also found that a significant amount of computation time was involved in calculating the composition of constraint relations. There are 2^{13} possible interval constraint relations, so caching the entire composition table ($64M$ entries) may not always be convenient. Also, our generic algorithm design should work with algebras with more than 13 atomic relations and therefore a much larger composition table. We thus compare various different methods of calculating the compositions. We observed that time performance of the algorithm design is strongly affected by the method of calculating compositions. In particular, the 'optimal method', caching the entire table, is so fast as to be 'off-scale'–we cannot effectively plot the relative-time results on the same graph as we plot times for the other methods. We conclude that, whenever possible, when extremely rapid solution is required, the entire composition table should be cached. The space-time trade-off is heavily in favor of time.

The computation of compositions takes place within path-consistency computations, but is not the only time bottleneck in these computations. Counting compositions as taking unit time, path-consistency is known to be cubic time [Mac77], and there is a class of examples on which iterative local algorithms must take quadratic time even on parallel machines [LM88, LM94]. However, there is some experimental evidence that iterative local algorithms generally take linear time in parallel [SHZ+91]. We compute path-consistency using the relational-matrix composition algorithm of [LM88], explained in Section 3. The speed of path-consistency computations is essential to the performance of the algorithm design, and the raw Ladkin-Maddux algorithm may be improved by tuning. For the larger consistent problems we generated, we found that path-consistency computations dominate solution time. We conclude that improving path-consistency computations is essential for good time performance of the constraint-solution algorithm design.

Large Consistent Problems We have also looked at solving consistent problems with large numbers (up to 500) of variables [RL92]. The techniques we found also 'scale down' to the smaller problems, so we discuss the techniques in this paper in the context of the smaller problems, without differentiating them. We summarise briefly here what we found by working on the larger problems. To solve large consistent problems, we used $C = pointisables$. Every constraint relation is a sum of very few pointisable relations, the average being ~ 3. Consequently, choosing pointisable subrelations during search instead of atomic subrelations reduces the search branching factor from an average of between 6 and 7 to an average of ~ 3. Space is the more significant limitation on solution algorithm performance rather than time, and it was possible to solve problems of 500 variables and $250,000$ non-trivial constraints using pointisable search, whereas problems of 200 variables caused significant memory problems with search using atomic subrelations. The maximal problem size is mainly restricted by the available memory space to hold the (reduced) network matrices in the depth-first search tree. We conclude that the branching factor during search is an important space performance factor which may be influenced by judicious choice of the class C.

These large problems were generated using the method of [RL92] (also noted in [LR92]), namely generating a configuration of intervals with the endpoints of each interval picked randomly from a finite collection of possible endpoints. This yields a solvable interval constraint problem with atomic constraints, and to each constraint noise was then added. At these larger sizes, using this problem generation method, it turns out that a path-consistency computation returns a solution by itself in one pass, and the solution time is expected $O(n^3)$. For networks of between 200 and 500 variables, the solution is found by the initial path-consistency computation. The ratio (average solution speed for k-variable networks): (average solution speed for m-variable networks) for $100 \leq m < k \leq 500$ is thus approximately $k^3 : m^3$. (The fact that solutions are often found on the first path-consistency pass for problems with over 200 variables may lead one to question this particular method for generating large consistent problems, but there is yet no more effective method available.)

In practice (e.g. Nökel's technical diagnosis system [Nök91]), large application problems are unlikely to exhibit the characteristics of random networks, and one may expect that domain specific knowledge can be exploited to aid in the solution process. However, domain-specific knowledge is unlikely to reduce the roughly cubic time needed to solve large problems, unless it can considerably reduce the time spent in path-consistency calculations.

Summary We knew already [LR92] that recurring path-consistency computations are an effective pruning method, that practically eliminates backtracking except in the 'phase transition' [Sel95] of interval constraint problems. We also knew that the phase transition for $c = 1$ lay between $6 \leq n \leq 15$, and that a majority of time in the algorithm was spent in path-consistency computations, in particular in composition calculations. In this paper, we show that

- the phase transition lies between $6 \leq c \cdot n \leq 15$;
- it is important to use larger classes C fulfilling the conditions (a)–(c) above, to reduce branching factor during search, and thus to solve problems with many more variables;
- expected behavior is cubic time on large problems;
- path-consistency calculations retain the greatest potential for speed-up;
- when composition is a table look-up, the algorithm is extremely fast;
- other methods of calculating composition still greatly improve over the 'naive method' of Allen;

The first four points were included in [LR93], the latter two results are new to this paper.

Recent Related Work. Nebel [Neb96] has evaluated the algorithm in this paper on a class of hard interval constraint problems within a different distribution of problems from the one which we consider. He looks at problem instances in the classes $A(n, d, s)$ in which n is the number of nodes (variables), d is the average degree of a node, and s is the average size of a constraint (*size* is the number of atoms that sum to form the constraint), the constraints selected from a uniform distribution. For fixed s, he identifies the range of values of d in which the phase transition occurs, and shows empirically that this range of values is roughly independent of n. (Our work here identifies a phase transition function which has n as an argument, for a different problem instance distribution.) Additionally, he considers the class $H(n, d)$ of problem instances with n and d as before, but in which a constraint on a pair of nodes is selected uniformly from a particular set of 3006 'probably very hard' constraints (based on theory from [NB95]). He also finds that using $C =$ *the ORD-Horn relations* yields some improvement in performance for some problems in the phase transition region over using $C =$ *the pointisable relations*, although the average performance remains roughly the same for both choices of C. The table of ORD-Horn compositions is 20 times larger than that for pointisables, and contains roughly 750,000 entries.

The techniques we describe in this paper are not limited to interval constraints - they work for any domain which is describable as a finite relation algebra. In principle, this includes all finite domains, although for an arbitrary finite domain the combinatorics of expressing it as a relation algebra might negate any advantage obtained by using our methods (a major difference is that in standard pair-search methods [Güs89] the relational operators are evaluated pointwise on the element pairs in the constraints as the algorithm progresses, whereas to use our algorithm the value returned by the operators on entire constraints must be known when required at a given point in the algorithm). Perhaps this point could be expressed by saying that we would expect our method to work efficiently for any domain which is *efficiently* describable as a finite relation algebra.

Other problem domains with efficient descriptions as relation algebras occur in spatial reasoning. The work of Randell, Cohn and Cui bases spatial reasoning on a class of "*jointly exhaustive and pairwise disjoint sets of binary relations for pairs of spatial regions*" [CRC95]. Such a class forms the atomic relations of a Boolean algebra of binary relations, and a relation algebra may be defined therefrom by calculating the composition of each pair of atoms (and also the converse of each atom, but this is usually straightforward if not trivial) [LM94]. Calculation of the composition is not always easy, for example for this so-called "RCC" theory of spatial relations [RCC92b]. The authors note that "*the task of constructing [composition] tables poses an interesting problem for automated theorem provers in its own right*" [RCC92a]. Bennett has improved the calculation for the RCC theory by using techniques based on propositional logics [Ben94]. Our algorithm architecture would function to solve constraint problems in the RCC theory of spatial relations, with $C =$ the atomic (RCC) relations. The compositions of relations in C would be calculated for use by the algorithm. However, fulfilling condition (b) for the RCC relations may require additional theorem-proving of the sort discussed by Randell, Cohn, Cui and Bennett.

Maddux has presented certain relation algebras called the *compass algebras* [Mad94], which can be interpreted as describing particular directions (say, when describing relative position) in n-dimensional Euclidean space. He discusses the (mathematical) solution of constraint problems over the compass algebras. Many are NP-complete, including those for relatively simple compass algebras: "*essentially all but the most trivial compass and interval algebras have NP-hard constraint satisfaction problems*" [Mad94]. Our algorithm could be used to solve compass algebra CSPs. Again, using $C =$ the set of atoms of the particular compass algebra, the compositions of atoms must be evaluated for use by the algorithm, and constraint problems involving only atomic constraints must be easily soluble (condition (b)).

Overview of the Paper. In Sections 2 and 3 we introduce the theoretical background for symbolic constraint problem solving. We describe the generic algorithm design in Section 4. Section 5 discusses ways of tuning the computation of path-consistency, including a comparison of alternative methods of calculating composition. Section 6 summarises previous experimental results, and presents new results on interval problems.

2 Binary Constraint Networks

We introduce binary constraint problems and their mathematical background, using interval constraint problems as an example. A binary constraint problem on n variables $x_1, ..., x_n$ is given by a formula

$$(P_{12}(x_1, x_2) \ \wedge \ P_{13}(x_1, x_3) \ \wedge \ ...... \ \wedge \ P_{n-1,n}(x_{n-1}, x_n))$$

where the P_{ij} are predicate symbols representing relational constraints on the variables x_i taken in pairs [Mac87]. Constraints are thus binary relations. A constraint problem is *satisfied* by finding values for the variables which satisfy the logical formula. Since we represent binary constraint problems as graphs (e.g. Figure 3), we shall also refer to a constraint problem as a *network* whose nodes are labelled with the variables and whose edges are labelled with symbols representing the constraint relations between the nodes.

In general constraint problems, the constraints P_{ij} may be given by explicit tuples of values, or by a rule which generates explicit tuples of values (e.g. [Güs89]). Alternatively, the constraints may be given symbolically. One way is for the constraints to be specified as *unions* of distinguished relations from some small class A. (Binary relations mathematically speaking are sets of pairs, and *union* is normal set union. We shall also call it a *sum*, to conform with relation-algebraic usage.)

The properties we require of this distinguished subclass A are that:

(i) relations in A are pairwise disjoint (no pair of values belongs to more than one relation);

(ii) the union of all members of A is everything (every pair of values belongs to one of these relations); and

(iii) all the constraint relations may be written as a union of relations in A.

We call members of a class A satisfying (i) - (iii) *atomic relations*. The class of all possible unions of the atomic relations forms a Boolean algebra, of which the atomic relations are atoms [BS81]. Further, if this Boolean algebra is closed under composition and converse, and contains the equality relation, then this class forms a relation algebra [JT52].

We use interval relations to illustrate. An *interval* is a pair $\langle a, b \rangle$ of real or rational numbers, such that $a < b$ [All83]. We consider just the binary relations on intervals obtained by considering the ordering (or equality) of their endpoints. There are 13 atomic relations in this class[2]. Seven of them are defined in Figure 1, and the other six are obtained by taking the *converses* (see below) of these relations[3]. The atomic relations are sets of interval pairs $\langle\langle x, y\rangle, \langle x', y'\rangle\rangle$. There are thus 2^{13} possible interval relations, obtained by taking all possible unions of the atomic relations[4]. By the connection between set-union and logical or ($\vee$), we may write a binary constraint in the logical form

$$P_{ij}(x_i, x_j) \ \equiv \ (\ (x_i \ R_1 \ x_j) \ \vee \ ....(x_i \ R_q \ x_j)\ )$$

[2] A general reference for interval relations is [vB91].

[3] In most papers, including [All83], these relations are defined by pictures.

[4] Note that although the relations themselves are infinite, there are only finitely many relations obtained by this procedure.

Equals:	$Id(S) = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x = x' < y = y' \in \mathbf{R}\}$
Precedes:	$P = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < y < x' < y' \in \mathbf{R}\}$
During:	$D = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x' < x < y < y' \in \mathbf{R}\}$
Overlaps:	$O = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < x' < y < y' \in \mathbf{R}\}$
Meets:	$M = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x < y = x' < y' \in \mathbf{R}\}$
Starts:	$S = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x = x' < y < y' \in \mathbf{R}\}$
Finishes:	$F = \{\langle\langle x, y\rangle, \langle x', y'\rangle\rangle : x' < x < y = y' \in \mathbf{R}\}$
Converses:	$P^{\smile}, D^{\smile}, O^{\smile}, M^{\smile}, S^{\smile}, F^{\smile}$

Figure 1: The Formal Definition of The Interval Relations

	$1'$	p	$p^{\smile}$	d	$d^{\smile}$	o	$o^{\smile}$	m	$m^{\smile}$	s	$s^{\smile}$	f	$f^{\smile}$
$1'$	$1'$	p	$p^{\smile}$	d	$d^{\smile}$	o	$o^{\smile}$	m	$m^{\smile}$	s	$s^{\smile}$	f	$f^{\smile}$
p	p	p	1	u	p	p	u	p	u	p	p	u	p
$p^{\smile}$	$p^{\smile}$	1	$p^{\smile}$	$v^{\smile}$	$p^{\smile}$	$v^{\smile}$	$p^{\smile}$	$v^{\smile}$	$p^{\smile}$	$v^{\smile}$	$p^{\smile}$	$p^{\smile}$	$p^{\smile}$
d	d	p	$p^{\smile}$	d	1	u	$v^{\smile}$	p	$p^{\smile}$	d	$v^{\smile}$	d	u
$d^{\smile}$	$d^{\smile}$	v	$u^{\smile}$	n	$d^{\smile}$	$z^{\smile}$	$y^{\smile}$	$z^{\smile}$	$y^{\smile}$	$z^{\smile}$	$d^{\smile}$	$y^{\smile}$	$d^{\smile}$
o	o	p	$u^{\smile}$	y	v	x	n	p	$y^{\smile}$	o	$z^{\smile}$	y	x
$o^{\smile}$	$o^{\smile}$	v	$p^{\smile}$	z	$u^{\smile}$	n	$x^{\smile}$	$z^{\smile}$	$p^{\smile}$	z	$x^{\smile}$	$o^{\smile}$	$y^{\smile}$
m	m	p	$u^{\smile}$	y	p	p	y	p	a	m	m	y	p
$m^{\smile}$	$m^{\smile}$	v	$p^{\smile}$	z	$p^{\smile}$	z	$p^{\smile}$	b	$p^{\smile}$	z	$p^{\smile}$	$m^{\smile}$	$m^{\smile}$
s	s	p	$p^{\smile}$	d	v	x	z	p	$m^{\smile}$	s	b	d	x
$s^{\smile}$	$s^{\smile}$	v	$p^{\smile}$	z	$d^{\smile}$	$z^{\smile}$	$o^{\smile}$	$z^{\smile}$	$m^{\smile}$	b	$s^{\smile}$	$o^{\smile}$	$d^{\smile}$
f	f	p	$p^{\smile}$	d	$u^{\smile}$	y	$x^{\smile}$	m	$p^{\smile}$	d	$x^{\smile}$	f	a
$f^{\smile}$	$f^{\smile}$	p	$u^{\smile}$	y	$d^{\smile}$	o	$y^{\smile}$	m	$y^{\smile}$	o	$d^{\smile}$	a	$f^{\smile}$

$x = (p + o + m)$
$y = (d + o + s)$
$z = (d + o^{\smile} + f)$
$a = (1' + f + f^{\smile})$
$b = (1' + s + s^{\smile})$
$u = (x + y) = (p + o + m + d + s)$
$v = (x + z^{\smile}) = (p + o + m + d^{\smile} + f^{\smile})$
$n = (z + z^{\smile} + b) = (y + y^{\smile} + a) = (1' + f + d + o + s + f^{\smile} + d^{\smile} + o^{\smile} + s^{\smile})$

Figure 2: The Relation Composition Table for the Interval Algebra

where the R_p are specific atomic relations from Figure 1. We thus see that an interval constraint problem is a logical formula in a restricted conjunctive normal form, in which each individual clause only has literals containing the same two free variables.

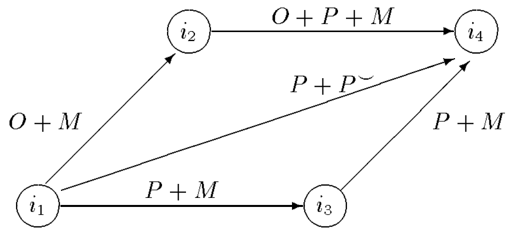
We shall write interval relations in infix form. Consider the example constraint problem

$$(i_1\, O\, i_2 \;\vee\; i_1\, M\, i_2) \;\wedge\; (i_1\, P\, i_3 \;\vee\; i_1\, M\, i_3) \;\wedge\; (i_1\, P\, i_4 \;\vee\; i_1\, \check{P}\, i_4) \;\wedge$$

$$(i_2\, O\, i_4 \;\vee\; i_2\, P\, i_4 \;\vee\; i_2\, M\, i_4) \;\wedge\; (i_3\, P\, i_4 \;\vee\; i_3\, M\, i_4)$$

shown in graphical form as a constraint network in Figure 3. The relation $P_{12}(i_1, i_2)$, for example, is $(i_1\, O\, i_2 \;\vee\; i_1\, M\, i_2)$.

Relation Sum. The two literals in the P_{12} clause share the same arguments (in the same order). The expression defines a binary relation between these two arguments, which is the set-theoretic union of the relations defined by each literal (since the logical connective is $\vee$).



$i_1(O + M)i_2 \;\wedge\; i_1(P + M)i_3 \;\wedge\; i_1(P + P^{\smile})i_4$
$\wedge\; i_2(O + P + M)i_4 \;\wedge\; i_3(P + M)i_4$

Figure 3: An Interval Constraint Problem in 4 variables and its Network

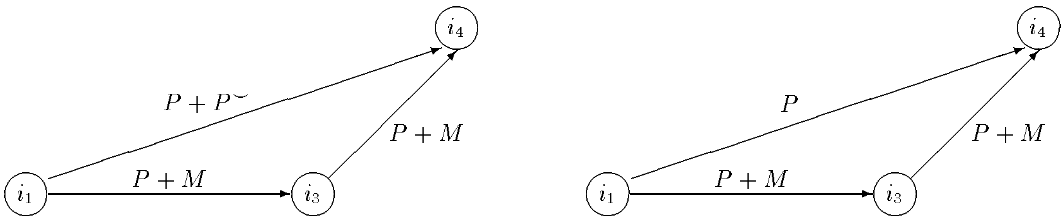
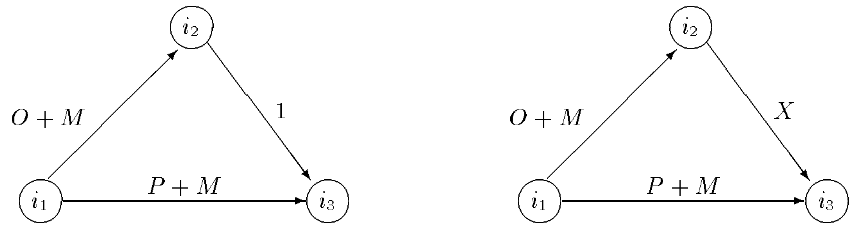


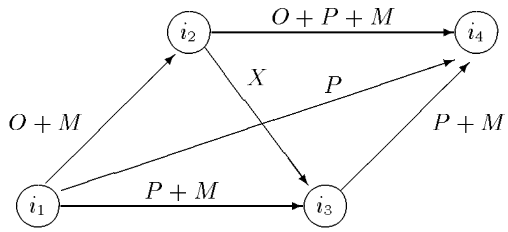
Figure 4: A Triangle Violating Path Consistency, and its Modification



$(O + M)^{\smile} \circ (P + M)$ is
$(1' + P + M + O + S + D^{\smile} + F^{\smile} + S^{\smile})$

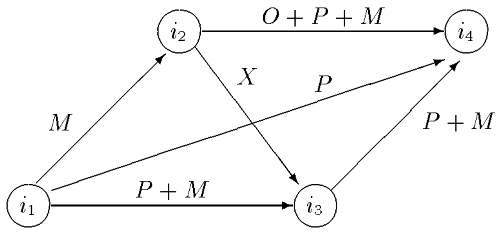
X is $(1' + P + M + O + S + D^{\smile} + F^{\smile} + S^{\smile})$

Figure 5: Another (Implicit) Violating Triangle and Its Modification



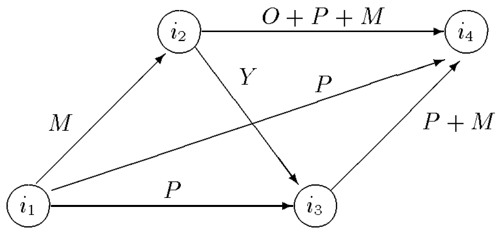
X is $(1' + P + M + O + S + D^{\smile} + F^{\smile} + S^{\smile})$

Figure 6: The Path-Consistent Reduction



X is $(1' + P + M + O + S + D^{\smile} + F^{\smile} + S^{\smile})$

Figure 7: Choice of M between i_1 and i_2



Y is $(P + M + O + D^{\smile} + F^{\smile})$

Figure 8: Choice of P between i_1 and i_3, and reduction

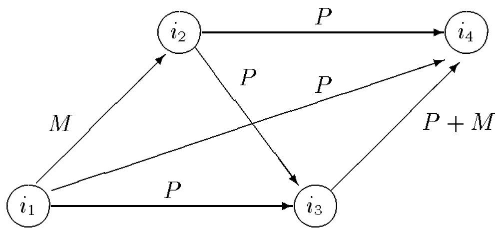


Figure 9: Choice of P between i_2 and i_3, and reduction

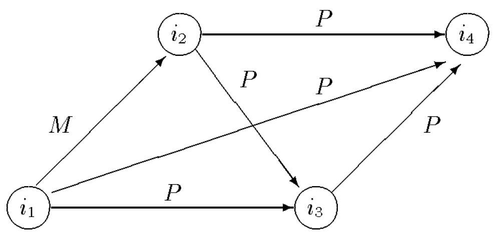


Figure 10: Final choice of P between i_3 and i_4

We denote this relation by $(O + M)$ and the corresponding operator on relations by $+$. We can now write the problem more compactly:

$$i_1 \,(O + M)\, i_2 \;\wedge\; i_1 \,(P + M)\, i_3 \;\wedge\; i_1 \,(P + \breve{P})\, i_4 \;\wedge\; i_2 \,(O + P + M)\, i_4 \;\wedge\; i_3 \,(P + M)\, i_4$$

One calculates the relation sum symbolically: it is easy to see that the sum of $(P+M+O)$ and $(O + S + F + M)$ is the sum of atoms occurring in either, i.e. $(P + M + O + S + F)$.

Relation Intersection. Relation intersection is just intersection of sets. The symbolic calculation of this operation is also simple, when relations are represented as sums of atoms. Atomic relations are pairwise disjoint (i.e. the intersection of any two of them is empty), thus trivial set theory shows that the intersection of two relations such as $(P + M + O)$ and $(O + S + F + M)$ is just the sum of the atomic relations common to both, namely $(O + M)$.

Relation Converse. The *converse* $\breve{R}$ of a relation R is defined by the equivalence

$$x\breve{R}y \;\equiv_{def}\; yRx$$

If the relation R labels the edge $[i, j]$ in a network, then the same information is conveyed by a label of $\breve{R}$ on the reverse edge $[j, i]$. The *converse* operation is needed in path-consistency computations.

Composition, or Relative Product. Given binary relations $R(x, y)$, $S(x, y)$, the *composition* $R \circ S$ of the relations is defined by

$$(R \circ S)(x, y) \Leftrightarrow (\exists z)(R(x, z) \;\&\; S(z, y))$$

Pointisable Relations. A *pointisable relation* on intervals is a relation which may be written as a conjunction of equality and inequality relations on its endpoints. For example, let x_L, x_R denote respectively the left and right endpoints of interval x. Then $x(O + M)y$ may be written as $(x_L < y_L \;\wedge\; y_L \leq x_R)$, and thus is pointisable. However, not all formulas may be written thus, e.g. $x(P + \breve{P})y$. The reader may verify that a succinct formula for this relation is the disjunction $(x_R < y_L \;\vee\; y_R < x_L)$. Interval constraint problems all of whose constraints are pointisable may be written without loss of information as constraint problems whose variables are the endpoints, and whose non-trivial, non-empty constraints are one of $<, =, >, \leq, \neq, \geq$. Such problems may be solved in quadratic time [vB90], and all such problems which are path-consistent are also soluble [LM88, LM94].

There are 187 pointisable relations, defined and investigated in [LM88], and independently in [vBC89]. Nökel's [Nök91] technical diagnosis system uses algorithms on 1-convex and 2-convex relations, which are subclasses of the pointisable relations. For further information on the relation between pointisable interval constraint problems and equivalent point networks, see [LM88, KL91].

3 3-Consistency and Path-Consistency

We here describe the property of path-consistency, which turns out to be equivalent to 3-consistency. We present a formal representation and an algorithmic formulation of a general path-consistency algorithm based on relational matrices.

The composition table of the 13 atomic relations on intervals was given in [All83], and shown in Figure 2, using auxiliary definitions of eight other (non-atomic) interval relations.

From the composition table from atoms, one calculates the composition of two arbitrary relations by using the distributive laws $R \circ (S+T) = (R \circ S) + (R \circ T)$ and $(R+S) \circ T = (R \circ T) + (S \circ T)$. Use of these distributive laws is very time-consuming, and can easily account for most of the computation time unless care is taken. We compare the various methods of calculating composition in Section 5. There is a space/time tradeoff in how much of the entire composition table to store or cache, and how much to compute. However, the tradeoff is weighted very heavily in favor of time–when the entire table is cached, our solution algorithm is extremely fast.

3-Consistency. By the definition, $(a,b) \in (R \circ S)$ if and only if there is a value c such that $(a,c) \in R$ and $(c,b) \in S$. For any satisfying values $a_1, ..., a_n$ for the variables $x_1, .., x_n$ we must have, by this definition, $\langle a_i, a_k \rangle \in P_{ik} \Rightarrow \langle a_i, a_k \rangle \in P_{ij} \circ P_{jk}$, for any $i, j, k \leq n$. This necessary condition may be used as a pruning technique to narrow down the potential choices of a_i and a_k: we calculate the relation $P_{ij} \circ P_{jk}$, and then intersect it with P_{ik} to form the new, potentially smaller relation (and thus tighter constraint) $P_{ik} \cap (P_{ij} \circ P_{jk})$. We then use this as the new constraint on x_i and x_k. We call this a *triangle operation.* A triangle operation on our example problem shown in Figure 3 is given in Figure 4, and another in Figure 5. We say that a triangle operation *stabilises* if the result is identical to the original label, equivalently that $(P_{ij} \circ P_{jk}) \subseteq P_{ik}$. We say that a network is *3-consistent*, or *is stable*, if every triangle operation stabilises. The stable network resulting from triangle operations on Figure 3 is shown in Figure 6. To stabilise a network, triangle operations must be performed on all possible constraints P_{ij} using all length-2 paths for every P_{ij} (which involves also using the *converse* operation on labels)[5]. In general, triangle operations must be iterated for each P_{ij} for the network to stabilise. 3-consistency is equivalent to path-consistency [Mon74], so we follow standard terminological abuse by conflating the two notions.

3-Consistency Is Relatively Cheap. 3-consistency is the least expensive form of consistency-checking for interval networks. *Arc-consistency* (2-consistency) is computationally cheaper, but unfortunately every interval problem is guaranteed to be arc-consistent [LM88].

Complexity. Path-consistency is a necessary condition for consistency but does not imply consistency [All83]. As noted previously, path-consistency algorithms are serial cubic time [Mac77], or parallel $O(n^2 \, log \, n)$ time, and iterative algorithms may take $\theta(n^2)$ time [LM88], but some studies conclude they can run in near linear time on fast parallel machines [SHZ+91, Sus91]. Path-consistency computations have been implemented in a planning system which uses interval computations [Koo88, Koo89].

A Path-Consistency Reduction Scheme. Denote *intersection* of relations by '$\cdot$', and *composition* of relations by '$\circ$'. Let P_{ij} be the relational constraint between variables x_i and x_j. A path-consistency computation may be regarded as computing the greatest fixed point of the following sets of equations (i.e. the collection of largest possible relations r_{ij} satisfying them):

$$r_{ij} \leq P_{ij}$$

$$r_{ij} = r_{ij} \cdot \prod_k (r_{ik} \ \circ \ r_{kj})$$

where the symbol $\prod_k$ denotes the product taken over all $k \leq n$. This fixed point is the most-general path-consistent reduction of the original network. This observation leads to the representation of constraint problems by $(n \times n)$ matrices of relation symbols, where the

[5] This is how all 3-consistency algorithms in the literature work.

```
function PC (var M: matrix): boolean;
    repeat
        M ← M · M²;
    until M = M²;
    return (M ≠ 0);
end;
```

Figure 11: Path-Consistency Algorithm

matrix M corresponding to the constraint problem A has entries $M_{ij} = P_{ij}$. We call such a matrix a *relational matrix*. By its nature, the matrix has to represent a complete graph, so it contains an entry of **1** (the universal relation[6] , i.e. the union of all 13 atoms) wherever no constraint edge appears in the original problem.

Product and *composition* for relational matrices are defined by the following schemes, which use intersection and composition defined on relations:

$$(M \cdot M')_{ij} = M_{ij} \cdot (M')_{ij}$$

$$(M \circ M')_{ij} = \prod_{k \leq n} M_{ik} \circ (M')_{kj}$$

Define also the power M^n by the usual induction, $M^1 = M$, and $M^{n+1} = (M^n) \circ M$. So, for example, $M^2 = (M \circ M)$, as we use in our path-consistency function in Figure 11. More information about relation matrices may be found in [LM94].

A Path-Consistency Algorithm. The path-consistency algorithms in the literature to date are all *iterative local* algorithms [LM94], differing mainly in the details of the iterations over the triangles. Our formulation in Figure 11 works on a relational matrix M, which is iteratively reduced in the **repeat** loop. If PC detects a zero edge in M it halts immediately and returns false, thereby indicating that the network is not path-consistent.

In a practical implementation, only the triangles of edges P_{ij} whose labels have changed in the previous iteration need to be recomputed. Some authors [Mac77, All83] propose a queue data structure for maintaining the triangles that must be recomputed. The computation then simply proceeds until the queue is empty. In our implementation we flip a bit in the network matrix, denoting that this element must be recomputed in the next iteration. This preserves the same edge evaluation order from one iteration to the next.

We say that a network *fails* if a path-consistency computation detects inconsistency, and it *succeeds* if it stabilises without detecting inconsistency. Succeeding networks are usually consistent, but this need not be always the case, as we show in Figure 16.

4 A Generic Solution Algorithm Architecture

Figure 12 illustrates our generic solution algorithm design as a flow chart. Details on the path-consistency check and the backtracking search are given in the pseudo-code in Figs. 11 and 13. At the core of any interval constraint solver is an intelligent backtracking search, with some pre- and postprocessing.

We found a path-consistency check effective for pre-processing, but other reduction schemes are possible (not arc-consistency!). Path-consistency is a necessary condition for consistency,

[6] Called 'no info' in [All83].

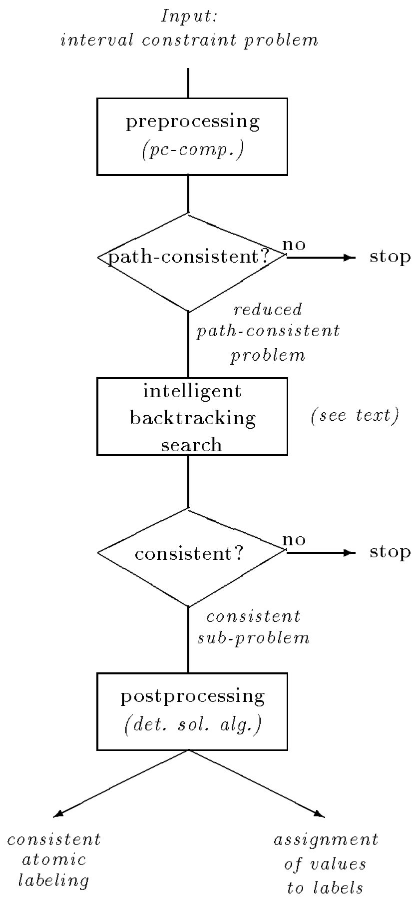


Figure 12: General Algorithm Design

and often suffices to prove a network inconsistent. Preprocessing with path-consistency has the additional advantage that it returns a logically equivalent, syntactically *reduced network*, in which many atoms that can not be part of a solution have been deleted. The reduction is more pronounced for large networks than for small ones, so that few or no nodes need to be expanded in the solution search.

After preprocessing, the network is further reduced by an intelligent backtracking search until it is known to be consistent. A network is *consistent* if there exists a solution, or equivalently for the case of interval problems, a consistent atomic reduction [LM88]. Figure 13 gives pseudo code for our consistency check routine, CC. It works on a relation matrix M and is started on the first edge M_{12}, i.e. function CC is initially invoked with parameters M, $i = 1$ and $j = 2$. CC selects a sub-label $k_l \in E$ from the edge M_{ij}, where E is (in our case) either the set of pointisable labels or the set of the 13 atomic labels. The path-consistent reduction of the resulting network is calculated, and if it is non-trivial (i.e. not inconsistent), CC recursively deepens on the next edge, and selects again a sub-label $l \in E$ on that edge. When all selected atomic labels form a path consistent network, the whole network is consistent [LM88]. If the network does not succeed at some point, the algorithm backtracks and chooses another sub-label l on that label. Only when all sub-labels have been unsuccessfully tried may the entire network be deemed inconsistent. Figures 7 - 10 show an actual run of the algorithm on the initial path-consistent reduction (Figure 6) of our example (Figure 3).

```
function CC (var M: matrix; i, j: integer): boolean;
{Recursive consistency computation, starting at edge M_ij}
    M' ← M;                                          {save matrix M}
    for each sublabel l_k ∈ M_ij do begin
        M_ij ← l_k;
        if PC(M) then
            if M_ij is last edge or CC(M, next_i, next_j) then
                return (true);
        M ← M';                                      {restore M}
    end;
    return (false);                                  {no consistent atomic labeling}
end;
```

Figure 13: Consistency Computation Algorithm

	p	$\breve{p}$	d	$\breve{d}$	o	$\breve{o}$	m	$\breve{m}$	s	$\breve{s}$	f	$\breve{f}$	$\sum$
p	1	13	5	1	1	5	1	5	1	1	5	1	41
$\breve{p}$	13	1	5	1	5	1	5	1	5	1	1	1	41
d	1	1	1	13	5	5	1	1	1	5	1	5	41
$\breve{d}$	5	5	9	1	3	3	3	3	3	1	3	1	41
o	1	5	3	5	3	9	1	3	1	3	3	3	41
$\breve{o}$	5	1	3	5	9	3	3	1	3	3	1	3	41
m	1	5	3	1	1	3	1	3	1	1	3	1	25
$\breve{m}$	5	1	3	1	3	1	3	1	3	1	1	1	25
s	1	1	1	5	3	3	1	1	1	3	1	3	25
$\breve{s}$	5	1	3	1	3	1	3	1	3	1	1	1	25
f	1	1	1	5	3	3	1	1	1	3	1	3	25
$\breve{f}$	1	5	3	1	1	3	1	3	1	1	3	1	25

Table 1: Number of Atoms resulting from Label Compositions

Each PC-call in the search reduces the network matrix M. The resulting search tree is bushy near the root and gets thinner in deeper levels. The larger the network, the more reductions are achieved, because in the large networks more label compositions are intersected, constraining the remaining label to a greater extent. In very large networks (eg. $n > 100$), the initial path-consistency check seems often to suffice to determine the result.

To obtain an actual solution, an assignment of values to variables, from a consistent atomic labelling, one needs to use van Beek's procedure of $O(n^2)$ steps, as mentioned.

5 Speeding Up Path Consistency

A complete network of size n has $n \cdot (n-1)/2$ edges. For every edge, the compositions of all $n-2$ alternative length-2-paths (the other possible edges of the triangle) must be intersected with the original edge, giving a total of

$$\frac{n \cdot (n-1) \cdot (n-2)}{2}$$

compositions and intersections. Computing the converse relations (the other triangular matrix) requires another $n \cdot (n-1)/2$ operations. For size 10 networks, only 405 operations are needed, but for size 50, we get 60,025 and for size 100 even 490,050 operations. To make things worse, path-consistency is an iterative process, that usually needs more than one iteration to stabilize. Path-consistency seems to stabilize on interval problems quite rapidly in practice (Figure 17). This figure also shows that larger random networks stabilize earlier than small ones. This convenient property is due to the larger number of intersections, which effectively reduces the size of the labels.

5.1 Improved Path-Consistency Algorithms.

The path-consistency algorithm of Section 3 simply recomputes all labels in every iteration. In practice, it suffices to recompute only the triangles of edges P_{ij} whose labels changed in the previous iteration. Some authors [Mac77, All83] proposed a queue data structure for maintaining the triangles that must be recomputed. The path-consistency computation then simply proceeds until the queue is empty. For our implementation, it turned out to be faster to flip a bit in the network matrix, denoting that this element must be recomputed next time. This also preserves the evaluation order for the next iteration. This method has also been used by van Beek [vBM96].

5.2 Implementing Triangle Operations

An efficient implementation of the triangle operation

$$new\text{-}P_{ij} = P_{ij} \cap (P_{ik} \circ P_{kj})$$

$(1 \leq i, j, k \leq n, i \neq j \neq k \neq i)$ must perform '$\cap$' and '$\circ$' as fast as possible. It seems natural to implement labels as bit vectors. Thus the fastest operations on a modern RISC processor (namely the bit-wise AND operation) can be used to perform the intersection operation.

5.2.1 Speeding Up Composition.

Speeding up the composition operation itself could require a little more thought: various methods are compared by their effect on the average algorithm running time in Figure 14. However, there is a clear winner: the optimal method, a table look-up, which yielded such comparatively fast results that it couldn't be plotted on the same curve as the other methods! The optimal method should therefore be used whenever possible (it requires 64MB storage for the table for interval problems–see below).

Allen's method [All83]: A composition of non-atoms is computed by looking up the compositions of atomic sublabels in a simple composition table of size 13 by 13 and summing the results (composition commutes with sum). For each atom combination, the Allen composition table contains the compositions with all other atoms. Compositions are performed by looping over all atoms in a label. Between one and $12 \times 12 = 144$ atomic compositions must be looked up ($1'$ is a left- and right-identity for composition, so we don't include it in the table) and combined with a bitwise inclusive OR. Allen's method appears poor when compared with the four methods below.

The Optimal Method: Given enough memory, the full $2^{13} \times 2^{13}$ composition table can be pre-computed and stored. At run-time, a single memory reference yields the composition of any two labels. (It may be thought that a complete composition table would require slightly fewer entries since any composition $R \circ eq$ or $eq \circ R$ is trivial. However, an equality-check must then be added to the code, slowing it down.)

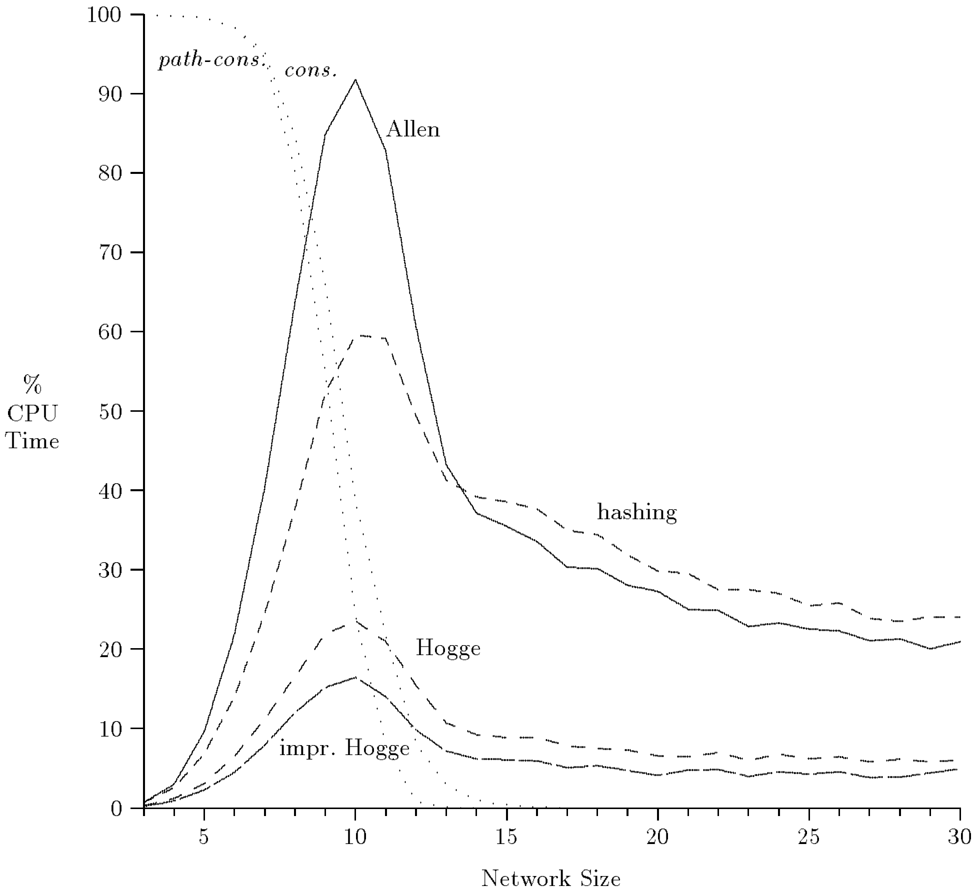


Figure 14: Runtime on complete networks, 1000 problems each

Implementation needs only half of that, because only an upper (or lower) triangular matrix must be built: the converse of a labeling is computed by flipping all 13 bits (= the number of atoms), since converse conjugates with composition. Since each entry has 2 bytes, one needs a total of 64 Megabytes memory for the table for interval constraint problems.

We experimented on a SUN SparcStation 20 with 128 MB main memory, with the upper triangular composition matrix (64 MB) held in memory. The program was found to be extremely fast: 10.000 problems each of size 3 to 30 took between 2 and 5 seconds only. It was far too fast for us to plot the results in Figure 14. It's clear that, when possible, the optimal method should be used.

5.2.2 Alternative methods

It might be infeasible for constraint problems in domains with more than 13 atoms to store the entire composition table. We thus consider three additional methods: one due to Hogge; an improvement of Hogge's method; and simple hashing. The results of using these different methods may be seen in Figure 14.

Hogge's method [Hog87]: John C. Hogge[7] devised a method that uses four tables instead of only one. These are of size $2^7 \cdot 2^7$ (bit 1-7 of rel1, bit 1-7 of rel2), $2^7 \cdot 2^6$ (bit 1-7 of rel1, bit

[7] We are grateful to Peter van Beek for pointing out Hogge's work to us.

8-13 of rel2), $2^6 \cdot 2^7$ (bit 8-13 of rel1, bit 1-7 of rel2), and $2^6 \cdot 2^6$ (bit 8-13 of rel1, bit 8-13 of rel2). The result is given by the union of the four array accesses plus 3 shift operations and some logical ANDs. A total of (16k + 8k + 8k + 4k) = 72k bytes memory space is needed.

An improved Hogge method : At the cost of a slight increase of memory space, Hogge's method can be improved to use only two tables, one holding the 7 low order atom compositions, and the other holding the 6 high order atom compositions. We need memory space of size $(2^{13} \cdot 2^7) \cdot 2$ bytes = 2 MByte plus $(2^{13} \cdot 2^6) \cdot 2$ bytes = 1 MByte. This amount of memory is even available on many Intel-based home PCs [8]. More important is the improved computation speed, requiring only two memory accesses, one logical shift and one OR operation.

Hashing: In consistent networks, path consistency computation is computed iteratively many times over the same triangles, while yielding only a few changes. This motivated us to implement a hashing table holding all previously computed compositions. Before a composition is computed, we look in the table to see whether it is there already. If so, we use it. If not, we compute the composition using Allen's method, store the result in the hash table and proceed. This hashing method has the advantage that any available amount of memory can be utilized. In our case, we used 2 MBytes.

Figure 14 shows that the hashing technique is effective in consistent networks, where the same edges may be evaluated in successive iterations. Hashing is faster (on a CPU-time basis) than Allen's method in complete networks of size < 14, where many iterations must be performed to prove a network consistent. An explanation of this effectiveness is given in Figure 15, which shows the percentage hit-rate in accessing the hash table, as a function of network size, on networks with $c = 1$. While the largest table of 2 mega-bytes gives best results, we can save 99% of the memory with only negligible performance reductions: A hashing table of only 8 kilo-bytes (2^{13}) gives almost the same results. In this table we simply used the exclusive or of the two 13-bit relations as a hashing index. Figure 14 also shows that reducing the table size further to only 256 bytes still gives significant savings over the naive composition method.

5.3 Improved Order of PC-Computations

One may consider ordering the edges of the constraint network in order to reduce the number of non-significant (or redundant) atoms and to stabilize faster.

Van Beek gives each edge a heuristic label and processes the edge-stack according to the heuristic value. He suggested the following criteria [vBM96]:

1. If either P_{ij} or P_{jk} is 1, then the result is 1.
2. If two edge-labels to be composed include respectively atoms P and $\breve{P}$; or $\breve{P}$ and P; or D and $\breve{D}$, then the result of the composition is 1. One may check these three conditions in the bit-vectors and conclude the result, without further calculation.
3. If, during calculation of the composition, it is determined that the composition is already larger than the label already constraining the target edge, the calculation for this edge-triple may be stopped, since it cannot further constrain the target edge : see [vBM96, Fig. 4].

In [RL92], we proposed some ordering schemes, namely:

[8] In fact, we did our implemented work on an Intel 80486 running the Linux operating system.

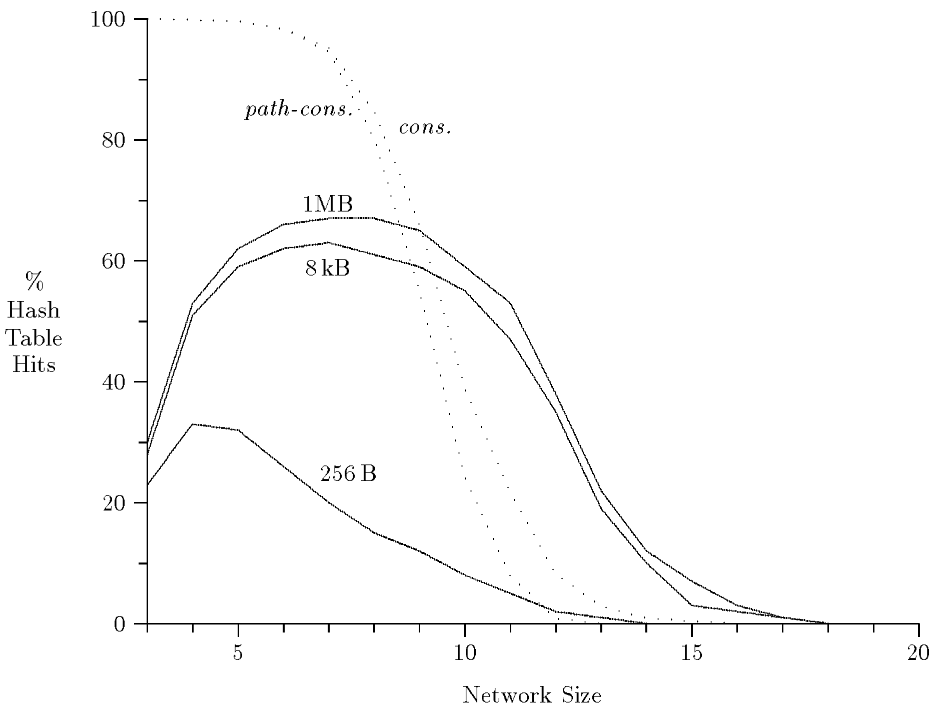


Figure 15: Hit rate for various hash table sizes (p=1.0, 1000 probs)

Skip atomic edges In the search process, skip all edges with atomic labels. Due to the previous path-consistency computation, atomic labels are known to be consistent with the other labels and hence do not further restrict the search space. This not only saves computing time, but more importantly saves memory space. For the network sizes we investigated, skipping atomic edges turns out to be almost essential.

Select $1'$-atoms first If an edge of the network contains an *equals*-relation, reduce the dimension of the network matrix by joining the two vertices and intersecting the labels of all adjacent edges. Formally, let P_{ij} contain a $1'$ (possibly among other atoms). Select the $1'$-relation first and perform for all $k \leq n$ the operations $P_{ik} \leftarrow P_{ik} \cap P_{jk}$ and $P_{jk} \leftarrow P_{jk} \cap P_{ik}$.

If $1'$ is the only atom of P_{ij}, delete the i-th row and the i-th column from the network matrix. The label intersection has already been computed in the preceding path-consistency computation.

Select point-meeting-relations first Select first those labels which have only few entries in the composition table: for example for those intervals starting or ending at exactly the same time instance: *equals, meets, starts, finishes* and their converses. This cuts the search space at the start by reducing the amount of atoms remaining in the network by what one hopes is a large amount. This is a standard technique in constraint satisfaction. In general, one would generate a statistic of label compositions giving the most reduction in the label size. This information could then be used to select the most probable compositions first. For the case of interval networks, ongoing research may determine the effectiveness of this strategy.

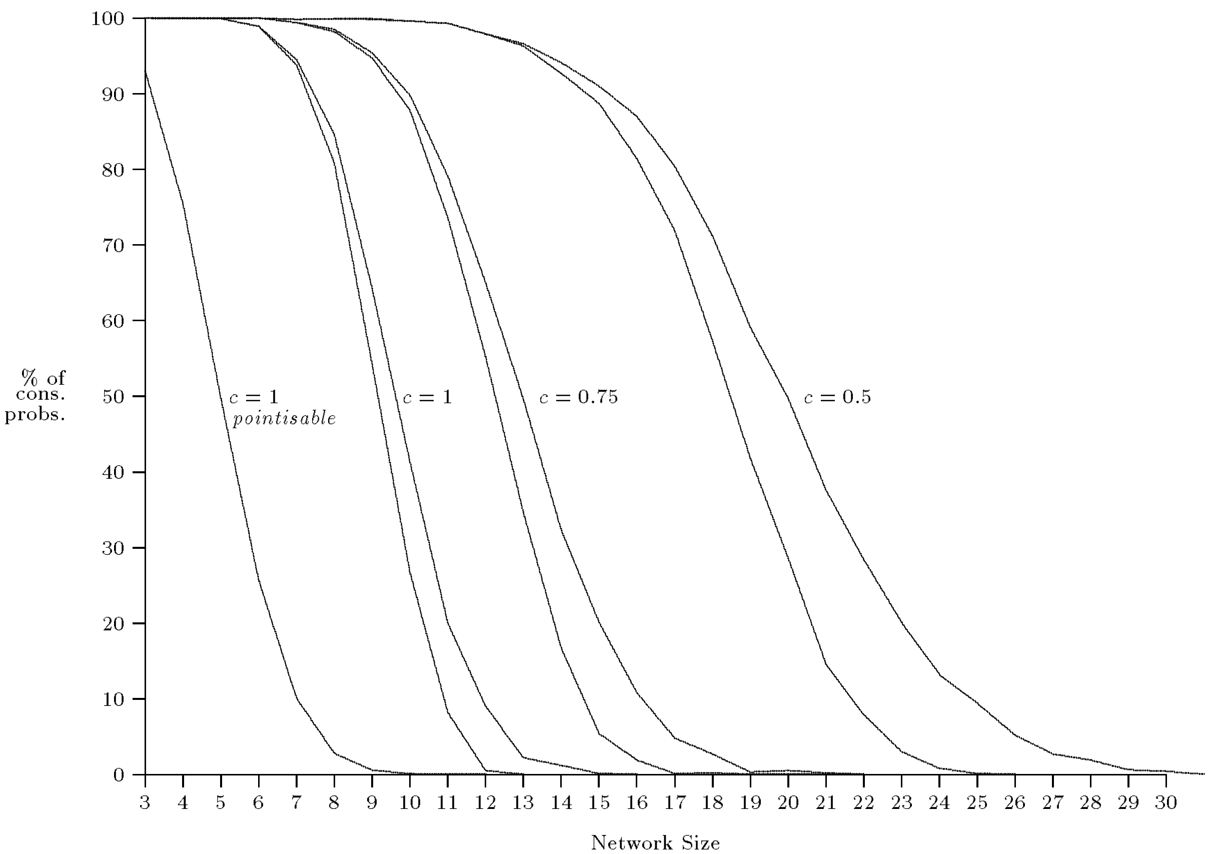


Figure 16: Percentage of Consistent Networks (1000 Probs.)

6 Empirical Studies of Random Networks

We generated interval constraint networks where the labels are independent, identically distributed random variables drawn from a discrete probability distribution. The networks have the following properties:

- Each atom within a label occurs with the same probability.
- Each label within the network occurs with the same probability.

We excluded initial networks with zero-labels, since they are known to be inconsistent. Also, since one cannot expect constraint graphs to be complete in practice, we generated graphs with different completion coefficients ranging from $c = 1$ (complete network) to $c = 0.5$ (every second edge has a non-trivial constraint). Unconstrained edges are labeled with the universal relation **1**, as explained above.

Figure 16 shows the percentage of networks of sizes $3 \leq n \leq 30$ that succeeded in the first path-consistency computation, and also those that were later found to be consistent by the solution search. Three pairs of graphs are shown (for completion coefficients of 1, 3/4 and 1/2). In each pair, the left hand graph gives the percentage of consistent networks and the right hand graph gives the percentage of path-consistent networks. For the pointisable networks, only one graph is shown, because here path-consistency implies consistency.

All graphs show a sharp decline in the consistency rate with increasing network size. We call it the *transition range*. For complete networks ($c = 1$), the transition range lies between

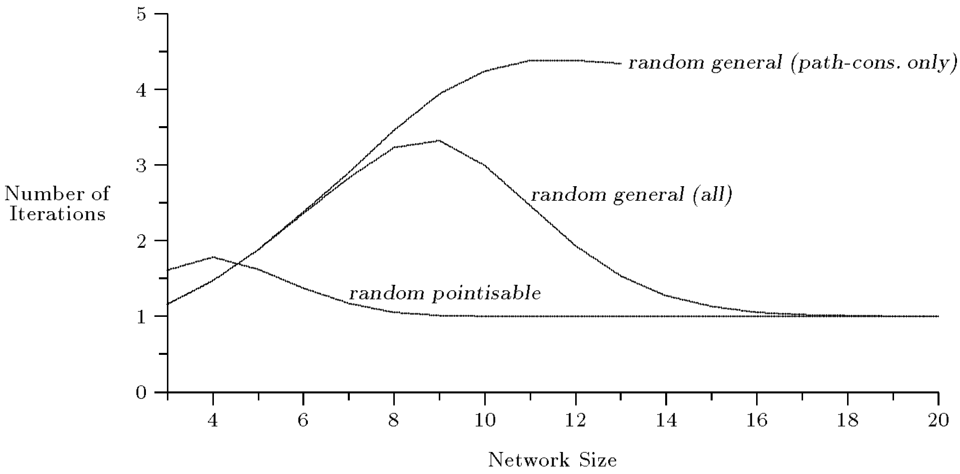


Figure 17: Iterations in Initial Path-Consistency Computation, 10,000 Probs.

the size 8 networks (where roughly 20% fail) and the size 11 networks (where roughly 80% fail). Networks in the transition range took particularly long to search, in comparison with networks outside the transition range. Even so, back in 1992, the average solution time required within the transition range was less than half a second of CPU time on a Sun SparcStation 1 (remember those?). Although the problem is NP-complete, we saw no case that needed longer than half a minute to solve, amongst the 200,000 complete networks generated.

Moreover, for $c = 1$, we found that *all* complete networks that we randomly generated with more than 13 variables are inconsistent, as are all pointisable networks with more than 11 variables [LR92]. While at sizes 14 to 17, some small number ($\leq 1\%$) of these inconsistent networks succeed, inconsistency of large networks is usually detected right in the initial path-consistency computation.

It is interesting to note that the transition range gets wider in the networks for which $c < 1$. Also the distance between the two graphs (consistent and path-consistent) is growing, which indicates that the initial path-consistency computation is not so effective in networks with smaller values of c. (We attribute this unwanted behavior to the smaller number of intersections of relations, which is less constraining.)

Average Number of Iterations to Stabilise. Figure 17 shows the average number of iterations required to stabilise in the path-consistency computation for complete networks. The number of iterations correlates with the transition range. The peak (of 3.3 iterations) occurs in the middle of the transition range at sizes 9 and 10.

While for sizes $n \leq 7$ always the same average number of iterations are required, regardless of whether a network is path-consistent or not, in larger sizes the consistent networks need more iterations to stabilise than the inconsistent ones. Inconsistency is usually detected right in the first iteration.

Initial Label Sizes Figure 18 gives the initial distribution of the label sizes in random networks. In absolute numbers, the white bars add up to $2^{13} = 8192$, which is the total number of atoms over all possible labelings, ranging from $\emptyset$ to $\mathbf{1}$. Here, the numbers are given relative to the maximum, $8192 = 100\%$.

In random networks, there are an average of $13/2 = 6.5$ atoms per label before the initial path-consistency computation. This would be the average branching degree in a brute force

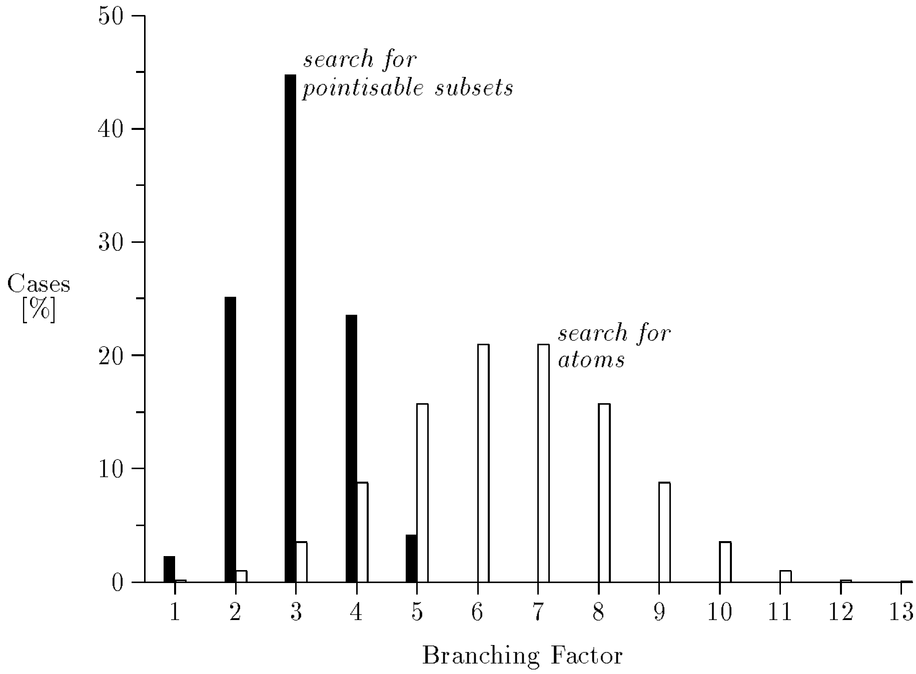


Figure 18: Distribution of Node Branchings

solution search when no path-consistency reduction is applied. The white bars in Fig. 18 show the distribution of the label sizes over the whole range from one to thirteen atoms. The curve has the shape of a Gaussian with the peak lying at 6 and 7, as it should be.

When searching for pointisable sub-labels, the branching is done on subsets rather than single atoms. This reduces the branching degree to an average of 3.02, as can be seen in the black bars in Fig. 18. Note that this gives a worst case estimate of the search branching degree, because the frequent path-consistency checks in (and before) the search further reduce the label size.

Since the branching degree is crucial to the overall performance of any search problem, we expect the search-for-pointisable-subsets to be much more efficient than a direct search for atomic labelings. In our experiments, we noted speedups between a factor of 3 and 7 for network sizes ranging from 10 to 50, respectively. In the very large problems of size $n > 100$, path-consistency is such an effective pruning technique that further search is only seldomly required. Hence, in the very large problems, the search-for-pointisable-subsets can not speed up solution time. However, as we have noted, it is nevertheless a significant factor in space efficiency.

Average Label Reductions by Path-Consistency. We also investigated the pruning power of the path-consistency computation in the search for atomic labelings. In inconsistent networks the first iteration of the path-consistency computation almost halves the label sizes, especially when the networks are very small (≤ 7) or very large (≥ 14). The following iterations then reduce the remaining atoms further, until a zero-label is found.

For the succeeding networks, also half of the total work is done in the first iteration. In an additional experiment with large (non-random) *consistent* networks, we found that path-consistency is more effective the larger the network. In the size-80 networks, for example, path consistency reduces the average label size to merely 1.1 atoms per label. In addition, fewer iterations are required. For the details of these experiments see [RL92].

Solution Density in the Search for Atomic Labelings. The effort needed to solve a given problem depends on the density of the solution nodes in the search space. As explained above, this effort is greatly reduced by the path-consistency computations, which reduces the size of the search tree. But how much does path-consistency reduce the average label size?

To answer this question, we modified our algorithm to examine the complete search space and return *all* solutions, rather than only one. Overall, we found a surprisingly high solution density. In networks of size $n < 8$ every 10th to 25th leaf node (path) is a solution. In the larger networks, more nodes must be searched to find a solution, with the peak lying at size 9-networks, where every $\approx$ 200th node contains a solution. But even in this worst case, the solution density is quite high considering the large search space.

When the path-consistency heuristic is applied in every interior node of the search tree (like in the above CC algorithm), even fewer nodes are expanded. The solution density is then at about 0.4, that is, there is a 40% probability for a node being a solution node. Unfortunately, the structure of the search tree is highly unbalanced, with the solution nodes clustering in certain subtrees. One cannot conclude that *each* interior search node has the same probability of lying on a solution path. But the given figure of 40% remains the same over the whole range of network sizes from $n = 3$ to $n = 11$.

Solution is Thus Expected Cubic Time. From the high solution density following each path-consistency computation, we can predict that the path-consistency computation dominates other aspects of the solution algorithm, and that in practice one can expect the number of path consistency computations to be almost constant. It therefore follows from [Mac77, MF85] that the solution procedure is *expected cubic time* for serial computations. Our statistics bear this out.

Resource Constraints. While for larger networks a cubic time solution algorithm could be expected to be impractical in terms of *time* taken, in our experience the excessive *space* requirements of the solution search are the more restrictive constraint. Even with a simple depth-first search approach the solution process builds a search tree of depth $d = n \cdot (n-1) \cdot 1/2$. For every interior node one (triangular) network matrix is stored on a stack, which results in a total space requirement of d^2 words. For size-100-networks with $c = 1$, for example, a maximum of 24 million memory words are needed. Note, however, that our program usually needs less storage than this, because it skips all atomic edges. In our experiments, we saw no case which needed a full search depth of $n \cdot (n-1) \cdot 1/2$.

7 Conclusions

We have presented a general algorithm architecture for solving binary constraint problems symbolically. Any class of constraint problems is included in a relation algebra, which, when finite, contains atomic relations. Our algorithm design may be used when there is a sub class C of the constraint relations for which

(a) the composition table may be stored and accessed efficiently;

(b) there is an efficient method for solving problems whose constraints are all in C;

(c) all constraint relations (including the atomic ones) may be represented as sums of elements of C.

The algorithm searches on subrelations of a constraint that are in C, and caches the composition table for all of C. This reduces the branching factor in search, which decreases the space requirements of the algorithm. It also reduces the time taken in calculating compositions of arbitrary constraints using the distributive law, but we suggest caching the entire table instead, when practical, rather than computing any of it.

Our experiments on interval problems showed that:

- Our methods are efficient enough to solve interval constraint problems of 500 variables and more in reasonable CPU time. This includes problems in the *transition range*, which has been observed by others typically to include the hardest areas of random problems [CKT91, Sel95].
- Path-consistency computations *during* the search are an excellent pruning technique, considerably reducing backtracking during search even on problems in the transition range. For problems outside the transition range, search becomes almost a linear selection of atomic (or pointisable) labels for the edges.
- The computation time is dominated by path-consistency computations, which are actual cubic time ($\approx 1/6 \cdot n^3$), in particular by calculating compositions of arbitrary constraint relations from the cached composition table.
- The computation space requirements are affected by the branching factor in the search, and are a more significant resource limitation than the time requirements. Hence, more sophisticated solution search methods (like best-first searches) are not generally needed.

We conclude that the critical performance factors in time and space are

- Reducing the branching factor during search (space critical);
- Caching all of the composition table, or if that cannot be done, using Hogge's method or hashing (a space/time tradeoff, heavily leaning towards time);
- Using a fast symbolic path-consistency algorithm (time critical).

Acknowledgements

This work has benefitted from continuing fruitful discussions with Peter van Beek, who has independently verified many of the phenomena. Peter also suggested experimenting with $C = pointisables$, and obtained the first results for this case, some of which appear in [vB92]. To him, our grateful thanks for ongoing cooperation.

References

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[Ben94] B. Bennett. Spatial reasoning with propositional logics. In *Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference (KR94)*, pages 51–62. Morgan Kaufmann, 1994.

[BS81] Stanley Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Springer Verlag, 1981.

[CKT91] P. Cheeseman, R. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–337, Sydney, Australia, 1991. Morgan Kaufmann.

[CRC95] A. G. Cohn, D. A. Randell, and Z. Cui. Taxonomies of logically defined qualitative spatial relations. *International Journal of Human-Computer Studies*, 43(5-6):831–846, 1995. Available from `http://agora.leeds.ac.uk/spacenet/publications.html`.

[DMP91] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, May 1991.

[Güs89] Hans-Werner Güsgen. *CONSAT: A System for Constraint Satisfaction.* Morgan Kaufmann/Pitman, 1989.

[Hog87] John C. Hogge. TPLAN: A temporal interval-based planner with novel extensions. Technical Report UIUCDCS-R-87-1367, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, September 1987.

[JT52] Bjarni Jònsson and Alfred Tarski. Boolean algebras with operators II. *American Journal of Mathematics*, 74:127–162, 1952.

[KL91] Henry A. Kautz and Peter B. Ladkin. Integrating metric and qualitative temporal reasoning. In *Proceedings of the 9th National Conference on AI (AAAI-91)*, pages 241–246, Anaheim, CA, 1991. AAAI Press.

[Koo88] Johannes A. G. M. Koomen. The TIMELOGIC temporal reasoning system. Technical Report 231, University of Rochester, Dept. of Computer Science, Rochester, NY, 1988.

[Koo89] Johannes A. G. M. Koomen. Localizing temporal constraint propagation. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR89)*, pages 198–202, Toronto, Canada, 1989. Morgan Kaufmann.

[Lad90] Peter B. Ladkin. Constraint reasoning with intervals: A tutorial, survey, and bibliography. Technical Report TR-90-059, International Computer Science Institute, Berkeley, CA, September 1990.

[LM88] Peter B. Ladkin and Roger D. Maddux. On binary constraint networks. Technical Report KES.U.88.8, Kestrel Institute, Palo Alto, CA, 1988. An extensively revised version appeared in [LM94].

[LM94] Peter B. Ladkin and Roger D. Maddux. On binary constraint problems. *Journal of the ACM*, 41(3):435–469, May 1994.

[LR92] Peter B. Ladkin and Alexander Reinefeld. Effective solution of qualitative interval constraint problems. *Artificial Intelligence*, 57(1):105–124, September 1992.

[LR93] Peter B. Ladkin and Alexander Reinefeld. A symbolic approach to interval constraint problems. In Jacques Calmet and John Campbell, editors, *Artificial Intelligence and Symbolic Mathematical Computing*, volume 737 of *Lecture Notes in Computer Science*, pages 65–84, Karlsruhe, Germany, 1993. Springer–Verlag.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Mac87] Alan K. Mackworth. Constraint satisfaction. In Stuart Shapiro, editor, *Encyclopedia of Artificial Intelligence*. Wiley Interscience, 1987.

[Mad94] Roger D. Maddux. Relation algebras for reasoning about time and space. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Algebraic Methodology and Software Technology, Enschede 1993*, Workshops in Computing Series, pages 27–44. Springer-Verlag, 1994.

[MF85] Alan K. Mackworth and Eugene C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25:65–74, 1985.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[NB95] Bernhard Nebel and Hans-Jürgen Bürckert. Reasoning about temporal relations: A maximal tractable subclass of Allen's Interval Algebra. *Journal of the ACM*, 42(1):43–66, January 1995.

[Neb96] Bernhard Nebel. Solving hard qualitative temporal reasoning problems: Evaluating the efficiency of using the ord-horn class. *Constraints*, 1996. To appear.

[Nök91] Klaus Nökel. *Temporally Distributed Symptoms in Technical Diagnosis*, volume 517 of *Lecture Notes in Artificial Intelligence*. Springer–Verlag, 1991.

[RCC92a] D. A. Randell, A. G. Cohn, and Z. Cui. Computing transitivity tables: a challenge for automated theorem provers. In D. Kapur, editor, *CADE-11, Proceedings of the 11th Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 786–790. Springer-Verlag, 1992. Available from `http://agora.leeds.ac.uk/spacenet/publications.html`.

[RCC92b] D. A. Randell, Z. Cui, and A. G. Cohn. A spatial logic based on regions and connection. In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3rd International Conference on Knowledge Representation and Reasoning, KR'92*, pages 165–176. Morgan Kaufmann, 1992. Available from `http://agora.leeds.ac.uk/spacenet/publications.html`.

[RL92] Alexander Reinefeld and Peter B. Ladkin. Fast solution of large interval constraint networks. In Janice Glasgow and Roland Hedley, editors, *Proceedings of the 9th Biennial Conference of the Canadian Society for Computational Studies of Intelligence (AI'92)*, pages 156–162, Vancouver, Canada, May 1992. Morgan Kaufmann.

[Sel95] Bart Selman. Stochastic search and phase transitions: AI meets Physics. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, volume 1, pages 998–1002, Montréal, Canada, August 1995. IJCAII.

[SHZ+91] Steve Susswein, Thomas C. Henderson, Joe Zachary, Chuck Hansen, Paul Hinker, and Gary Marsden. Parallel path consistency. Technical Report UUCS-91-010, Univ. of Utah, July 91.

[Sus91] Steven Susswein. Parallel path consistency. Master's thesis, University of Utah, Dept. of Comp. Sci., 1991.

[vB90] Peter G. van Beek. Reasoning about qualitative temporal information. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90)*, pages 728–734, Boston, MA, 1990. Morgan Kaufmann.

[vB91] Johan F. A. K. van Benthem. *The Logic of Time*. D. Reidel, second edition, 1991.

[vB92] Peter G. van Beek. Reasoning about qualitative temporal information. *Artificial Intelligence*, 58:297–326, 1992.

[vBC89] Peter G. van Beek and Robin Cohen. Approximation algorithms for temporal reasoning. In *Proceedings of the 11th Joint Conference on Artifical Intelligence (IJCAI-89)*, pages 1291–1296, Detroit, MI, 1989. Morgan Kaufmann. Long version appeared in [vBC90].

[vBC90] Peter G. van Beek and Robin Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–144, 1990.

[vBM96] Peter G. van Beek and Dennis W. Manchak. The design and an experimental analysis of algorithms for temporal reasoning. *Journal of Artificial Intelligence Research*, 4:1–18, 1996.

[VKvB89] Marc Vilain, Henry A. Kautz, and Peter G. van Beek. Constraint propagation algorithms for temporal reasoning. In Daniel Weld and Johan de Kleer, editors, *Readings in Qualititative Reasoning About Physical Systems*, pages 373–381. Morgan Kaufmann, San Mateo, CA, 1989.