Working Notes of the 1991 Spring Symposium on Constraint-Based Reasoning

Rina Dechter

September 16, 1991

Technical Report 91–65

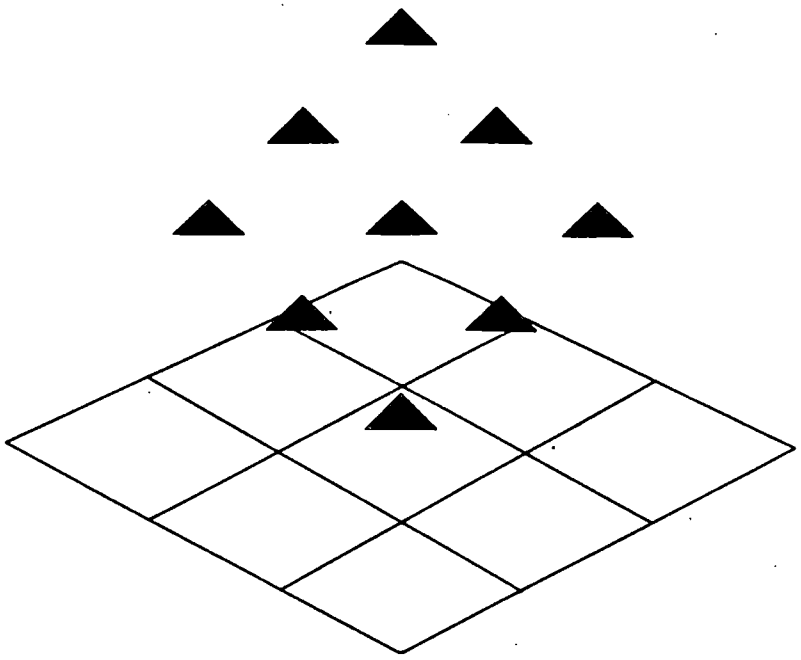
1991 Spring Symposium Series

Constraint-Based Reasoning

Working Notes
(Distribution limited to symposium attendees)

March 26 – 28, 1991
Stanford University

Sponsored by the
American Association for Artificial Intelligence



WORKING NOTES

AAAI
SPRING SYMPOSIUM SERIES

Symposium:
Constraint-Based Reasoning

Program Committee:
Rina Dechter, UC Irvine, Cochair
Sanjay Mittal, Metaphor Computer Systems, Cochair
Alan Borning, University of Washington
Mark Fox, Carnegie Mellon University
Eugene Freuder, University of New Hampshire
Joxan Jaffar, IBM T.J. Watson Research Center
Alan Mackworth, University of British Columbia
Judea Pearl, UC Los Angeles

MARCH 26, 27, 28, 1991

STANFORD UNIVERSITY

Table of Contents

Distributed and Parallel Constraint Satisfaction

Theoretical Issues-2

Constraint Logic Programming-2

FOREWORD

Constraint-based reasoning (CBR) is a paradigm that unifies many traditional areas in Artificial Intelligence. Simply stated, CBR encourages the formulation of knowledge in terms of a set of constraints on some entities, without specifying methods for satisfying such constraints. Many techniques for finding partial or complete solutions to constraint expressions have been developed, and have been successfully applied to tasks such as design, diagnosis, truth maintenance, scheduling, spatio-temporal reasoning, and user interface.

The symposium brought together a diverse group of researchers, and the work presented spanned many topics, from basic research and theoretical foundation to practical applications in industrial settings. It became apparent that from a mathematical viewpoint, the field has reached a certain level of maturity; algorithmic breakthroughs were not reported nor were they expected. On the other hand, most of the talks focused on *strengthening* CBR with new implementation tools or *extending* the technology to new areas of application.

The symposium opened with Alan Mackworth's overview of the interplay between constraint-based reasoning and various logical frameworks. It stressed the point that although constraint satisfaction problems can be expressed in other logical frameworks, the relational language provides a convenient means of encoding knowledge which often invites unique opportunities for efficient processing techniques.

The discussions that followed fell into four major categories: 1. Extensions to commonsense reasoning (default, causal, qualitative, temporal). 2. Parallel and distributed approaches, 3. Constraint-logic-programming languages, and 4. New application areas (layout design, natural languages, DNA analysis, mechanical design).

1. The session on commonsense-reasoning centered around issues in temporal reasoning. In particular, two approaches for combining quantitative temporal specification (e.g., metric networks) and qualitative specification (e.g., Allen's interval algebra) were presented and compared. Meiri (UCLA) treats points and interval as temporal objects of equal status, admitting both qualitative or quantitative relationships. Ladkin (ICSI) and Kautz (Bell Labs) maintain the qualitative and quantitative components in two separate subsystems, and provide sound rules for transforming information between the two.

In the area of default reasoning, Ben-Eliyahu (UCLA) and Dechter presented a new tractable class of default theories based on CBR mapping, and Freuder (UNH) discussed an extension of the constraint language that expresses imprecise knowledge. In qualitative reasoning, Kuipers (UT) reviewed issues in qualitative simulation and the role of constraint processing. This area concluded with Pearl's (UCLA) presentation of causal constraint networks– a new tractable class of constraint problem that utilizes the efficiency and modularity inherent to causal organizations.

2. Distributed and neural architectures for constraint processing received much attention. Kasif (John Hopkins) opened this discussion by surveying the theoretical aspects of parallel computations. We learned that constraint-satisfaction, and even arc-consistency are "non-parallelisable" (unless the network has no cycles), meaning that it is unlikely to be solved by polynomial number of processors in polylogarithmic time. However, linear speedup or

good average parallel time are still feasible. Following this survey, several distributed models were presented, all addressing the basic questions of whether constraint satisfaction can be achieved with neural-like computations, that is, whether constraints can be expressed as global minima of neural networks and whether these global minima can be approached by local computations.

We heard answers to some of these questions: Pinkas (Washington University) showed that any set of constraints, binary or non-binary, can be described as a Hopfield net, such that its global minima coincide with the set of solutions. Collin (Technion) and Dechter showed that even if we upgrade the computation power of the neurons to finite state automata, the constraint problem can be solved only by making one processor distinguished (an almost uniform model), or if the topology is a tree. Guesgen (German National Research Center) concluded this topic by presenting a uniform neural network algorithm that achieves a global solution at the expense of increasing memory requirement.

3. Two sessions were devoted to constraint logic programming (CLP). These languages integrate constraint satisfaction and operations research techniques within the logic programming paradigm. Jaffar (IBM) opened the discussion on this topic by describing the management of hard constraints in CLP systems. The idea is to delay the evaluation of non-linear constraints until (and if) they become linear, at which point they can be solved by efficient algorithms designed for this task. We heard from a group at West Advanced Technology (Abdullah, Epstein, Lim, and Freeman) how non-linear constraints can be managed using available packages like MATHEMATICA. Van Hentenryck (Brown) described improvements to arc-consistency algorithms for functional or monotone constraints, and discussed their relevance to the CHIP programming language. Wilson, Borning, and Freeman-Benson (UW) showed how solutions to CLPs can be obtained using hierarchical weighing of constraints and introduced control knowledge via imperative constraint programming.

4. Application domains were presented throughout the symposium. In the area of natural language processing, Haddock (Hewlett-Packard, U.K) described the use of consistency algorithms to solve noun phrase reference. A group at Schlumberger (Kramer, Pabon, Keirouz, and Young) developed a polynomial algorithm for solving geometric constraint satisfaction problems. Baykan and Fox (CMU) introduced disjunctive constraints that can conveniently deal with applications of job shop scheduling and floor plan layout, and Yap (IBM) showed how the restriction site mapping in molecular biology can be expressed as dynamic constraint satisfaction.

A central issue that was raised repeatedly was the need for having a standard set of large, representative, real life benchmarks for evaluating different constraint processing techniques.

Rina Dechter

The Logic of Constraint Satisfaction

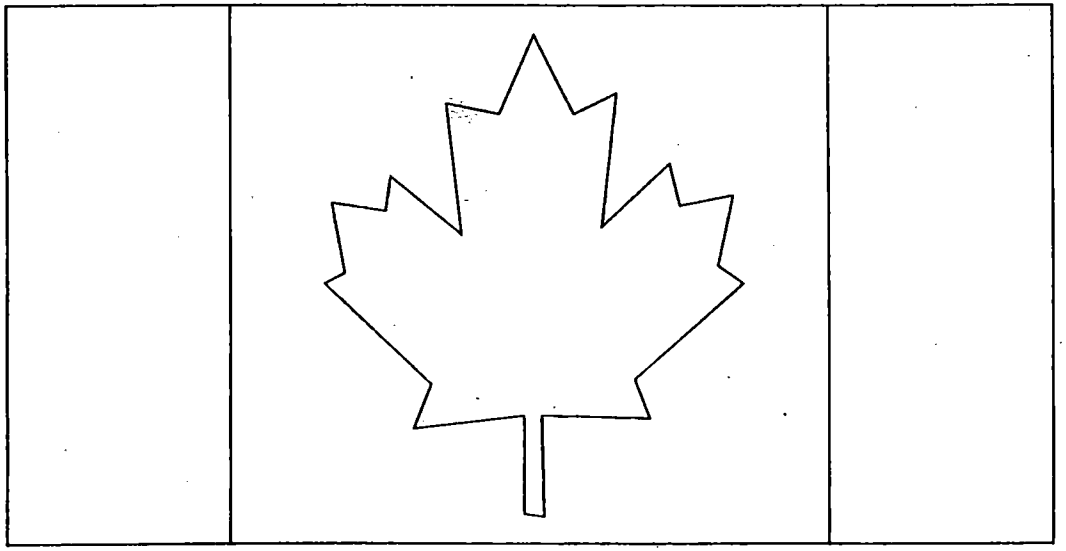
Alan Mackworth

Finite Constraint Satisfaction Problems in Logical Frameworks

- ☐ Can FCSP be posed in logical frameworks?
- ☐ Can standard logical methods be used to solve FCSP?
- ☐ Can properties of FCSP be exploited to get better algorithms?
- ☐ Are there tractable classes of FCSP?
- ☐ Do old results fall out?
- ☐ Can the logical FCSP approaches be generalized to CSP?
- ☐ Do we get new results & systems?

An FCSP: The Canadian Flag Problem

Colour the Canadian flag. Only two colours, red and white, should be used, each region should be a different colour from its neighbours, and the maple leaf should be red.



FCS as Theorem Proving in FOPC

FCSP decision problem: $Constraints \vdash Query?$

where $Query$: $\exists x_1 \exists x_2 \ldots \exists x_n QMatrix(x_1, x_2, \ldots x_n)$

$$\exists x_1 \exists x_2 \ldots \exists x_n P_{x_1}(x_1) \wedge P_{x_2}(x_2) \wedge \ldots \wedge P_{x_n}(x_n)$$
$$\wedge P_{x_1 x_2}(x_1, x_2) \wedge P_{x_1 x_3}(x_1, x_3) \wedge \ldots$$
$$\wedge P_{x_1 x_2 x_3}(x_1, x_2, x_3) \wedge \ldots$$
$$\vdots$$
$$\wedge P_{x_1 x_2 x_3 \ldots x_n}(x_1, x_2, x_3, \ldots, x_n)$$

$Constraints$ is a set of positive ground literals specifying the extensions of the predicates:

$$Constraints = \left\{ P_{x_{i_1} x_{i_2} \ldots x_{i_m}}(c_{i_1}, c_{i_2}, \ldots, c_{i_m}) | 1 \leq i_k < i_{k+1} \leq n \right\}$$

FCSP Decision Problem

A Finite Constraint Satisfaction Problem: $(Constraints, Query)$.

The decision problem — solution can be shown to exist or not:
$Constraints \vdash Query$, or $Constraints \nvdash Query$.

Proposition: Any FCSP $(Constraints, Query)$ is decidable.

Proof:

The Herbrand universe of the theory $Constraints \bigcup \neg Query$ is:

$$H = \{c \mid P(\ldots, c, \ldots) \in Constraints\}$$

H is finite.

FCSP Decision Algorithm

Decision Algorithm DA :

Success $\leftarrow$ No

For each $(x_1, x_2, \ldots, x_n) \in H^n$

If $Constraints \vdash QMatrix\,(x_1, \ldots)$ then **Success** $\leftarrow$ Yes

Report **Success**.

End DA

where $Constraints \vdash QMatrix\,(x_1, \ldots)$
if $\forall Atom \in QMatrix\,(x_1, \ldots)\ Atom \in Constraints$.

DA always terminates (in $O(|H|^n)$time).

The algorithm reports Yes iff $Constraints \vdash Query$. It reports No iff $Constraints \nvdash Query$.

Completing the constraints

Consider the completion of $Constraints$ with respect to $Query$. Each predicate can be completed (Clark,'78) in the following sense:

$$completion(Constraints) =$$
$$Constraints \cup \{\neg P_X(c_1, c_2, \ldots) | c_j \in H, P_X(c_1, c_2, \ldots) \notin Constraints\}$$

In other words, the complete extension of each predicate over the Herbrand Universe is specified in $completion(Constraints)$.

$Constraints \vdash Query$ iff $completion(Constraints) \vdash Query$.

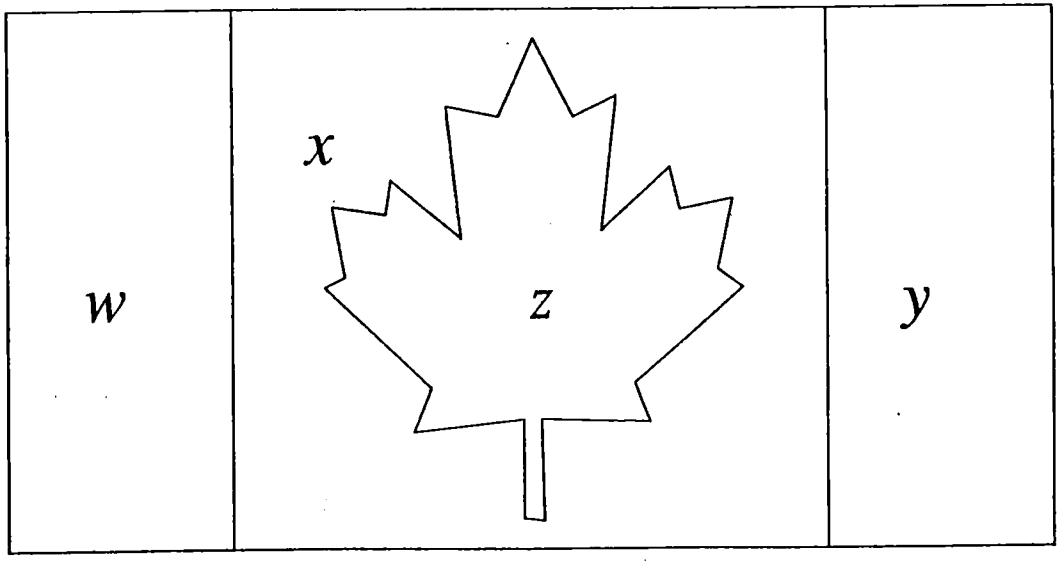
$Constraints \nvdash Query$ iff $completion(Constraints) \vdash \neg Query$.

Hence DA reports Yes iff $completion(Constraints) \vdash Query$ and No iff $completion(Constraints) \vdash \neg Query$.

Thus, we may choose to interpret the answer from DA in the original sense or under the assumption that the $Constraints$ have been completed. Both are correct.

The Flag FCSP in FOPC

Using the FCSP formalism presented above we can formulate the flag problem as follows.



Query :

$\exists w \exists x \exists y \exists z P(w) \wedge Q(x) \wedge R(y) \wedge S(z) \wedge N(w,x) \wedge N(x,y) \wedge N(x,z)$

Constraints :

$\{P(a), P(b), Q(a), Q(b), R(a), R(b), S(b), N(a,b), N(b,a)\}$

$H = \{a, b\}$ a stands for White, b for Red.

$H^4 = \{(a,a,a,a), (a,a,a,b), \ldots, (b,b,b,b)\}$

On the FCSP $(Query, Constraints)$ algorithm DA returns "Yes" succeeding on $\{w = b, x = a, y = b, z = b\}$.

CANADA

Logical Representation Systems

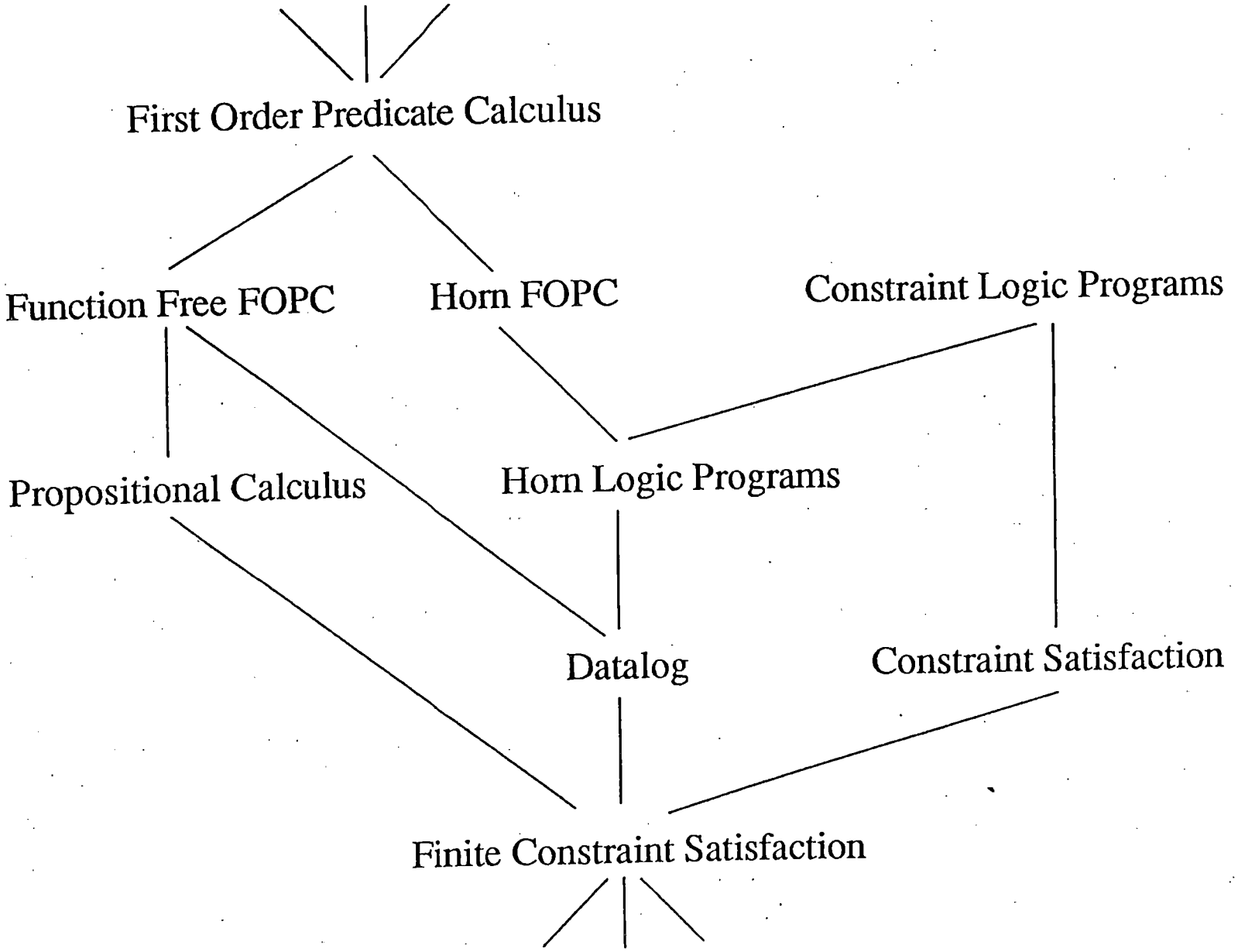
Faced with a problem, spectrum of logical representation systems.

Descriptive and procedural adequacy criteria — in conflict.

KISS principle: choose the simplest system.

FCS equivalent to theorem-proving in a very restricted form of FOPC.

- ☐ Horn FOPC restricts FOPC in only allowing Horn clauses, with at most one positive literal.
- ☐ Horn Logic Programs (HLP), with predicate completion, restrict Horn FOPC by allowing only one negative clause.
- ☐ Datalog restricts HLP by not allowing function symbols.
- ☐ FCS restricts Datalog by not allowing rules.


First Order Predicate Calculus
Function Free FOPC
Horn FOPC
Constraint Logic Programs
Propositional Calculus
Horn Logic Programs
Datalog
Constraint Satisfaction
Finite Constraint Satisfaction

FCS as Theorem Proving in Propositional Calculus

DA implements FCS as theorem proving in the propositional calculus.

$Query$ is a theorem?

$Constraints \bigcup \neg Query$ leads to a contradiction?

$$\neg Query : \neg \exists x_1 \exists x_2 \ldots \exists x_n QMatrix(x_1, \ldots)$$
$$\forall x_1 \forall x_2 \ldots \forall x_n \neg QMatrix(x_1, \ldots x_n)$$

$Constraints \bigcup \neg Query$ has no (Herbrand) models?

No $\forall$'s in $Query$ and so no $\exists$'s in $Constraints \bigcup \neg Query$.

Hence no Skolem functions in clausal form.

Herbrand universe is finite.

Replace $\forall$'s by the conjunction of the $\neg QMatrix(x_1, x_2, \ldots, x_n)$ clauses instantiated over H^n.

$$\{P(a), P(b), Q(a), Q(b), R(a), R(b), S(b), N(a,b), N(b,a)\}$$

$$\bigcup$$

$$\{\neg P(a) \vee \neg Q(a) \vee \neg R(a) \vee \neg S(a) \vee \neg N(a,a) \vee \neg N(a,a) \vee \neg N(a,a),$$

$$\neg P(a) \vee \neg Q(a) \vee \neg R(a) \vee \neg S(b) \vee \neg N(a,a) \vee \neg N(a,a) \vee \neg N(a,b),$$

.

.

$$\neg P(b) \vee \neg Q(a) \vee \neg R(b) \vee \neg S(b) \vee \neg N(b,a) \vee \neg N(a,b) \vee \neg N(a,b), \ [*]$$

.

.

$$\neg P(b) \vee \neg Q(b) \vee \neg R(b) \vee \neg S(b) \vee \neg N(b,b) \vee \neg N(b,b) \vee \neg N(b,b), \}$$

Propositional formula in CNF has a particular form: only unit positive clauses (arising from the constraints) and negative clauses (from the query). Horn.

Unsatisfiable iff the FCSP has a solution.

HornSAT — linear time but $|H|^n$ negative clauses.

Unit resolution alone is complete. Repeated unit resolution on $[*]$ reduces it to $\square$, corresponding to the solution $\{w = b, x = a, y = b, z = b\}$.

The HornSAT algorithm exactly mimics the algorithm DA.

The propositional variable in each unit positive literal is set to T and each negative clause is checked: if any clause has each (negative) literal required to be F then the formula is unsatisfiable otherwise it is satisfiable.

FCS as Theorem Proving in Horn FOPC

None so far serious candidates for actually solving a CSP: clarify the semantics and methods of the serious candidates.

Prolog interpreter: SLD-resolution is a sound, complete and somewhat efficient solution method.

```
%prolog
| ?- [user].
| p(a). p(b). q(a). q(b). r(a). r(b). s(b).
| n(a,b). n(b,a).
yes
| ?- p(W),q(X),r(Y),s(Z),n(W,X),n(X,Y),n(X,Z).
W = Y = Z = b,
X = a ;
no
| ?-
```

Checks every possible set of bindings for W, X, Y and Z.

By permuting the query one may reduce the size of the search space: a partially completed set of bindings can be rejected by a single failure. e.g. for the query:

```
p(W),q(X),n(W,X),r(Y),n(X,Y),s(Z),n(X,Z)
```

Heuristics, such as instantiating the most constrained variable next, can be used to re-order the query but, on realistic problems, this tactic is doomed.

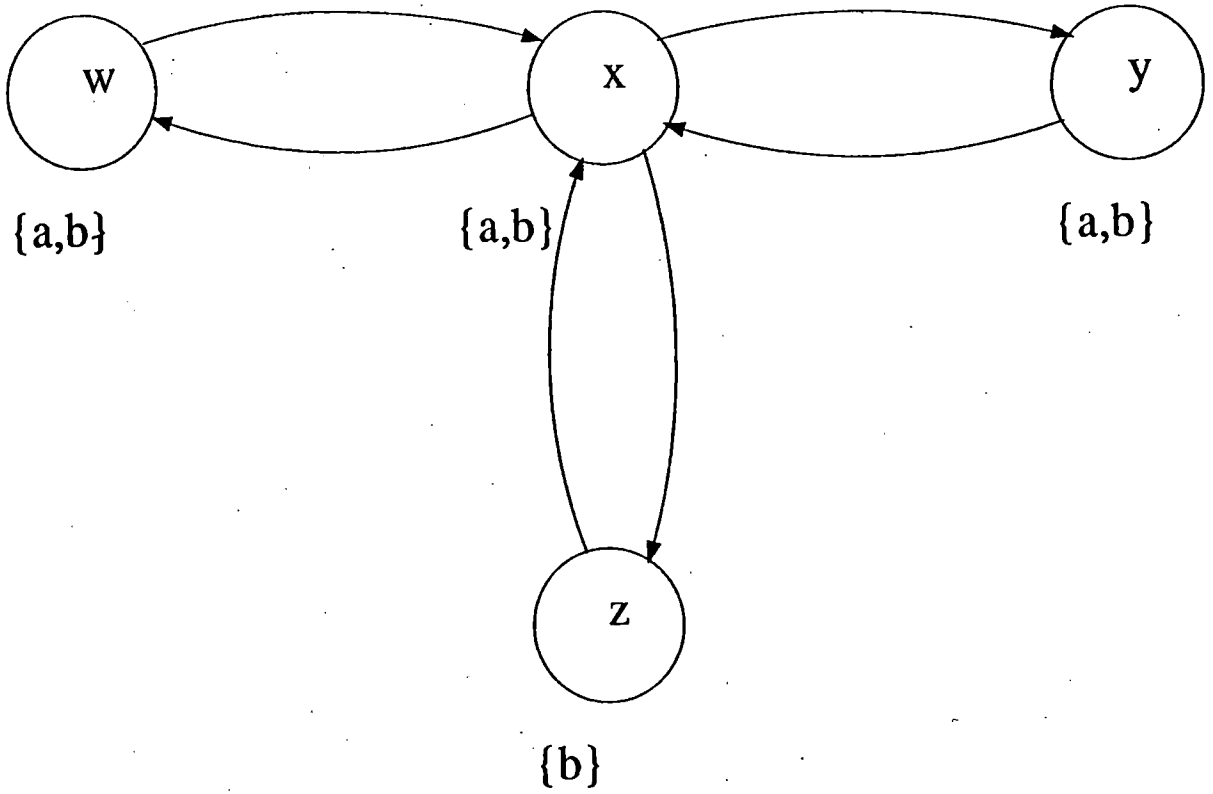
In general, no variable ordering can avoid *thrashing* by repeatedly rediscovering incompatible variable bindings.

FCS in Constraint Networks

Drawbacks of the SLD-resolution approach lead to FCS in *constraint networks*.

A constraint network represents each variable in the query as a vertex.

The unary constraint $P_x(x)$ establishes the domain of x, and each binary constraint $P_{xy}(x, y)$ is represented as the edge (x, y), composed of arc (x, y) and arc (y, x).



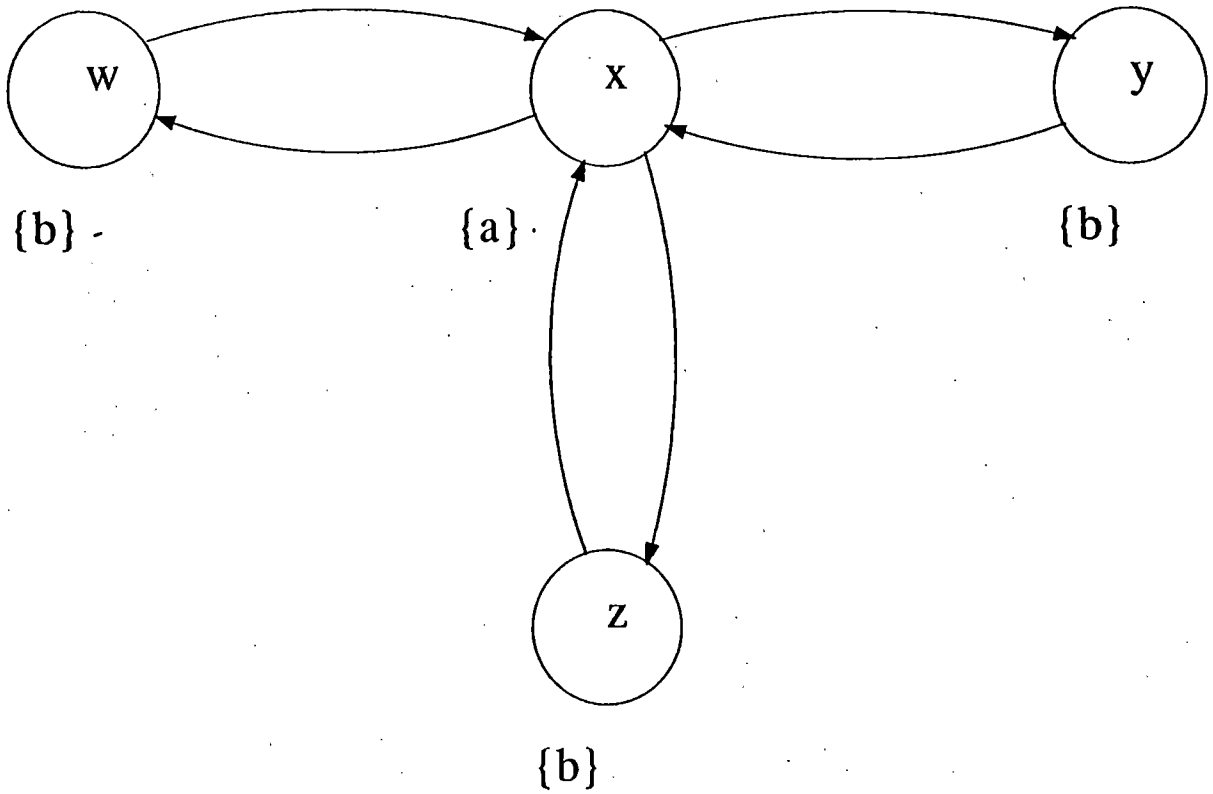
Constraint Network for the Flag Problem

An arc (x, y) is *consistent* iff

$$\forall u P_x(u) \rightarrow \exists v[P_y(v) \wedge P_{xy}(u, v)]$$

A network is arc consistent if all its arcs are consistent.

An arc (x, y) may be made consistent.



Arc Consistent Network for the Flag Problem

Arc consistency linear time.

If constraint graph a tree then arc consistency alone a decision procedure for FCSP (Mackworth & Freuder, '85).

Various other graph theoretic properties of the constraint network can be used to characterize and solve FCSPs (Freuder, '90; Dechter, '91).

An Arc Consistency Interpreter for FCSP

Given

Query :

$\exists w \exists x \exists y \exists z P(w) \wedge Q(x) \wedge R(y) \wedge S(z) \wedge N(w,x) \wedge N(x,y) \wedge N(x,z)$

Represent *Constraints* as:

$$P(w) \equiv w = a \vee w = b$$
$$Q(x) \equiv x = a \vee x = b$$
$$R(y) \equiv y = a \vee y = b$$
$$S(z) \equiv z = b$$
$$N(s,t) \equiv (s = a \wedge t = b) \vee (s = b \wedge t = a)$$

Rewrite *Constraints* using the AC rewrite rule:

$$P_x(u) \Leftarrow P_x(u) \wedge \exists v[P_y(v) \wedge P_{xy}(u,v)]$$

Here:

$Q(x)$

$\Leftarrow Q(x) \wedge \exists z[S(z) \wedge N(x,z)]$

$\Leftarrow (x = a \vee x = b) \wedge \exists z[z = b \wedge (x = a \wedge z = b \vee x = b \wedge z = a)]$

$\Leftarrow (x = a)$

Iterating the AC rewrite rule reduces the constraints to a fixpoint:

$$P(w) \equiv w = b$$
$$Q(x) \equiv x = a$$
$$R(y) \equiv y = b$$
$$S(z) \equiv z = b$$
$$N(s,t) \equiv (s = a \wedge t = b) \vee (s = b \wedge t = a)$$

In general, the interpreter must interleave the AC relaxation with some non-deterministic choice (Mackworth, '77). CHIP (Van Hentenryck, '87).

Connection Graphs (Kowalski, '75) with SLD-resolution on FCSP queries.

[F]CS and CLP($\mathfrak{D}$)

The FCS constraint form is a special case of the CLP($\mathfrak{D}$) rule form:

$$P(x, y, \ldots) \leftarrow C_1(x, y, \ldots) \wedge C_2(x, y, \ldots) \wedge \ldots$$
$$\wedge P_1(x, y, \ldots) \wedge P_2(x, y, \ldots) \wedge \ldots$$

In Horn Logic Programming $\mathfrak{D} = H$ and the C_i constraints are equalities on terms.

CS fits the CLP($\mathfrak{D}$) scheme e.g. CLP($\mathfrak{R}$).

$$P(x) \leftarrow (1 < x) \wedge (x < 3)$$
$$Q(y) \leftarrow (0 < y) \wedge (y < 2)$$
$$R(x, y) \leftarrow x < y$$

Refine:

$$P(x) \leftarrow (1 < x) \wedge (x < 2)$$
$$Q(y) \leftarrow (1 < y) \wedge (y < 2)$$

FCS as Model Finding in Propositional Logic

A radically different framework for FCS is as model finding in propositional logic (Reiter & Mackworth, '87; de Kleer, '89).

A formula F in propositional logic is constructed for the FCSP.

Each model of F corresponds to a solution of the FCSP.

Each proposition in F represents a possible binding of a variable to a value. $w : a$ means that variable w takes the value a.

F may be in CNF with a set of clauses representing:

- the fact that each variable must take a value (e.g. $w : a \vee w : b$)
- the fact that the values are pairwise exclusive (e.g. $\neg w : a \vee \neg w : b$)
- the constraints on related variables.

The constraints may be encoded as clauses in any suitable fashion. A *negative* encoding (de Kleer, '89) represents only the forbidden tuples of the constraints (e.g. $\neg w : a \vee \neg x : a$).

$$
\begin{aligned}
F = \{ & w : a \vee w : b, x : a \vee x : b, y : a \vee y : b, z : b, \\
& \neg w : a \vee \neg w : b, \neg x : a \vee \neg x : b, \neg y : a \vee \neg y : b, \\
& \neg w : a \vee \neg x : a, \neg w : b \vee \neg x : b, \neg x : a \vee \neg y : a, \\
& \neg x : b \vee \neg y : b, \neg x : b \vee \neg z : b \}
\end{aligned}
$$

SAT problem again.

In the propositional proof-finding framework the FCSP has a solution iff the formula has no models.

Under the model-finding framework each solution corresponds to a model of F.

Davis-Putnam? But SAT problem has the same special form again: no mixed clauses in this encoding.

Simplify the formula before deciding if there are any models.

Use two inference rules: negative hyperresolution, H_2, and unit resolution, U.

$$
\begin{array}{ll}
H_2: & p \vee q \vee r \vee \ldots \vee u \\
 & \neg p \vee \neg v \\
 & \neg q \vee \neg v \\
 & \neg r \vee \neg v \\
 & \ldots \\
 & \underline{\neg u \vee \neg v \qquad} \\
 & \neg v \\
U: & p \vee q \vee r \vee \ldots \vee u \\
 & \underline{\neg q \qquad\qquad} \\
 & p \vee r \vee \ldots \vee u
\end{array}
$$

Subsumption rules: S_p and S_n.

Given two positive clauses C_1 and C_2 where all the literals in C_1 appear in C_2 then S_p deletes C_2.

AC-resolution strategy: $(H_2\, S_n^*\, U\, S_p)^*$

$w : a \vee w : b$ $\qquad$ $w : b$

$x : a \vee x : b$ $\qquad$ $x : a$

$y : a \vee y : b$ $\qquad$ $y : b$

$z : b$

$\neg w : a \vee \neg w : b$

$\neg x : a \vee \neg x : b$

$\neg y : a \vee \neg y : b$

$\neg w : a \vee \neg x : a$ $\qquad$ $\neg w : a$

$\neg w : b \vee \neg x : b$

$\neg x : a \vee \neg y : a$ $\qquad$ $\neg y : a$

$\neg x : b \vee \neg y : b$

$\neg x : b \vee \neg z : b$ $\qquad$ $\neg x : b$

The simplified formula is:

$$F_s = \{w : b, x : a, y : b, z : b, \neg w : a, \neg x : b, \neg y : a\}$$

Exactly one model.

AC-resolution has the following properties:

1. The set of models of F is invariant.
2. No mixed clauses are generated so the division into positive and negative clauses is invariant.
3. Total number and length of clauses decreases monotonically.
4. $O(e)$.
5. 'Incomplete' — must be interleaved with search (e.g. assigning a truth value to a proposition) to decide the satisfiability of F.
6. AC-resolution, used here for model-finding, mimics behaviour of AC interpreter on FCSP (& Connection Graph theorem prover). Each proposition reifies possible substitutions for a variable.

This framework shows the relevance of planar SAT results (Seidel, '81) and 2SAT (only 2 values/variable — linear time) (Dechter).

N.B. Other encodings are possible. Directed constraint networks (Dechter and Pearl, '90) give Horn clauses for the constraints and hence HornSAT (linear time) if the input variables are given specific values.

Conclusions

- ☐ Can FCSP be posed in logical frameworks?
- ☐ Can standard logical methods be used to solve FCSP?
- ☐ Can properties of FCSP be exploited to get better algorithms?
- ☐ Are there tractable classes of FCSP?
- ☐ Do old results fall out?
- ☐ Can the logical FCSP approaches be generalized to CSP?
- ☐ Do we get new results & systems?

AAAI 1991 Spring Symposium Series
Constraint-Based Reasoning Symposium

Constraint-Based Knowledge Acquisition
Extended Abstract

Eugene C. Freuder
Computer Science Department
University of New Hampshire
Durham, NH 03824 USA
ecf@cs.unh.edu

Introduction

Constraint networks have begun to take their place as an AI knowledge representation paradigm alongside rules, frames, etc. [Guesgen, Junker, Voss, 87], and, in particular, as knowledge structures for knowledge-based systems [Havens and Rehfuss, 89]. This raises the issue of knowledge acquisition, which has been extensively studied in the context of other knowledge representation schemes and especially for rule-based expert systems.

I will discuss three projects that I have been involved with that explore problems and opportunities that arise in addressing knowledge acquisition issues in a constraint-based context. The first, carried out with Sue Huard [Huard, 90], assists an expert in *debugging* a constraint network knowledge base, in the spirit of Teiresias [Davis, 82]. The second, carried out with Suresh Subramanian [Subramanian and Freuder, 90], implements the *compilation* of rules from the observation of constraint-based problem solving, in the spirit of ACT [Anderson, 83]. The third project [Freuder, 86] proposes *induction* of constraint networks from examples in the spirit of ARCH [Winston, 75].

Debugging

There are several dimensions along which the debugging problem can be categorized:

1. The problem exhibited: erroneous solutions, missing solutions.
2. The nature of the bug: missing constraints elements, erroneous constraint elements.
3. User contribution: supply parts of missing solutions, identify erroneous solutions, recognize correct solutions, verify proposed missing or erroneous constraint elements.

We have primarily explored a single, simple point in this space of debugging contexts:

1. There is a single missing constraint element (specifying that a single pair of values is consistent).
2. This leads to a missing solution.
3. The user can supply a single value from the missing solution ("I know there is a solution where variable X has value a"), and can verify or reject solutions

proposed by the debugger.

This debugging context was explored for the n-queens problem, a very specific, but oft-studied CSP domain. The debugging problem was viewed as a partial constraint satisfaction problem [Freuder 1989]. Partial CSP algorithms were brought to bear. Heuristics were developed and compared for improving performance in two areas:

1. Program effort, using the usual measure of constraint checks, with the aim of providing quick feedback to the user.
2. User effort, measured by the number of questions asked of the user.

Compilation

We have developed a prototype system to compile rule-based knowledge from constraint-based problem solving experience. (For another form of CSP learning from experience see [Dechter, 86].) A simple constraint-based expert system solves problems. Its experience motivates the automated formation of rules, which can then be employed by a simple rule-based expert system. The rule-based system can function as a preprocessor to promote efficiency, or ultimately as a stand-alone system. There are two particularly interesting ways of viewing this process:

1. The constraint network implicitly contains many rules. By extracting previously successful reasoning steps in the form of explicit rules we provide a mechanism for offering heuristic guidance, learned from experience, to a brute force relaxation process.
2. We can attempt to compile out different specialized, efficient rule-based systems from a large knowledge base that is conveniently expressed initially in declarative, constraint-based terms.

These ideas have been tested on a knowledge base of New Hampshire birds, compiled with the assistance of a local wildlife biologist.

Induction

One of the most heavily studied forms of learning involves generalizing from examples. A well-known instance is Winston's ARCH program for learning structural descriptions of objects. Winston's program built semantic network descriptions. Using *constraint networks* instead we can take advantage of constraint-based contextual knowledge to leverage the learning behavior through constraint-based inference.

A simple example will illustrate. Winston's program learns that a wedge-shaped block supported by a brick-shaped block is a "house" structure. A "counterexample" showing the wedge and brick side by side teaches the program that the support relationship is required. Later another counterexample, showing one wedge on top of another, teaches the program that the bottom object cannot be a wedge. But is the second counterexample really necessary? (Indeed it seems a little odd; it could not be constructed physically.) If we view the support relationship as a constraint, a form of constraint-based reasoning (arc

consistency) allows a program to *infer* that the bottom object cannot be a wedge: wedges do not support bricks. This added intelligence on the part of the "pupil" lessens the burden on the "teacher" and speeds the learning process.

Acknowledgements: This material is based in part upon work supported by the National Science Foundation under Grant No. IRI-8913040. The Government has certain rights in this material. The author is currently a Visiting Scientist at the MIT Artificial Intelligence Laboratory.

References

[Anderson, 83] Acquisition of proof skills in geometry, in *Machine Learning*, Michalski, Carbonell and Mitchell, eds., Tioga Publishing Company, Palo Alto, Ca.

[Davis, 82] Teiresias: Applications of meta-level knowledge, in *Knowledge-Based Systems in Artificial Intelligence*, Davis and Lenat, McGraw-Hill, New York.

[Dechter, 86] Learning while searching in constraint-satisfaction-problems, *Proc. AAAI-86*, vol. 1.

[Freuder, 86] Applying constraint satisfaction search techniques to concept learning, Univ. of New Hampshire Tech. Rept. 86-33, Durham, NH.

[Freuder, 89] Partial constraint satisfaction, *Proc. IJCAI-89*, vol. 1.

[Guesgen, Junker, Voss, 87] Constraints in a hybrid knowledge representation system, *Proc. IJCAI-87*, vol. 1.

[Havens and Rehfuss, 89] Platypus: a constraint-based reasoning system, *Proc. IJCAI-89*, vol. 1.

[Huard, 90] Debugging Erroneously Overconstrained Constraint Networks, Ms. Thesis, University of New Hampshire, Durham, NH.

[Subramanian and Freuder, 1990] Rule compilation from constraint-based problem solving, *Proceedings of the 2nd International IEEE Conference on Tools for Artificial Intelligence.*

[Winston, 75] Learning structural descriptions from examples, in *The Psychology of Computer Vision*, Winston, ed., McGraw-Hill, New York.

Default Logic, Propositional Logic and Constraints

Rachel Ben-Eliyahu
< rachel@cs.ucla.edu >

Cognitive Systems Laboratory
Computer Science Department
University of California
Los-Angeles, California 90024

Rina Dechter
< dechter@ics.uci.edu >

Information & Computer Science
University of California
Irvine, California, 92717

January 29, 1991

Abstract

We present a mapping from a class of default theories to sentences in propositional logic, such that each model of the latter corresponds to an extension of the former. Using this mapping we show that many properties of default theories can be determined by solving propositional satisfiability. In particular, we show how CSP techniques can be used to identify, analyze and solve tractable subsets of Reiter's default logic.

Topic: Automated Reasoning / constraint satisfaction

Secondary Topic: General Knowledge Representation / Default Reasoning

1 Introduction

Since the introduction of Reiter's default logic ([Rei80]), many researchers have elaborated its semantics ([Eth87], [Kon88], [BF88]) and have developed inference algorithms for various default theories ([Eth87]),[KS89], [Sti90]). As it was clear from the beginning, most of those computations are formidable (not even semi-decidable), and research in the last few years, has focused on restricted classes of the language, searching for tractable subclasses of default theories. Unfortunately, many simplified sublanguages still remained intractable ([KS89], [Sti90]).

Since Reiter's logic is an important formalism for nonmonotonic reasoning, it is worth exploring new dimensions along which tractable classes can be identified. The approach we propose here examines the structural features of the knowledge base, and leads to a topological characterization of non-monotonic theories.

One language that has received a thorough topological analysis is *constraint networks*. This propositional language, based on multi-valued variables and relational constraints is also intractable, but many of its tractable subclasses have been identified by topological analysis. A constraint network is a graph (or hypergraph) in which nodes represent variables and arcs represent pairs (or sets) of variables that are included in a common constraint. The topology of such a network uncovers opportunities for problem decomposition techniques and provides estimates of the problem complexity prior to actual processing. Graphical parameters such as the *width* and the *cycle-cutset*, were identified as crucially related the problems' difficulty, and lead to effective solution strategies ([Fre82]), [MF84], [DP89]).

Our approach is to identify tractable classes of default theories by mapping them into tractable classes of constraint networks. Specifically, we reformulate a default theory within the constraint network language and, use the latter to induce the appropriate solution strategies.

Rather than attempting a direct translation to constraint network, this paper describes an intermediate translation of a class default theories into *propositional logic*. Since, propositional logic can be translated into constraint networks (and vice versa [dK89]), this yields a mapping to constraint networks. The intermediate translation into propositional logic may point to additional tractable classes and can shed new light on the semantics of defaults theories.

The bulk of this paper is devoted to the analysis of such transformation. We show that any disjunction-free propositional default theory with semi-normal rules can be translated, in polynomial time, to a set of propositional sentences , and, moreover, all the interesting properties of the default theory can be computed by solving the satisfiability of the latter. We then show how constraint networks can be utilized to identify tractable classes of default theories.

The paper is organized as follows. Sections 2 and 3 describe Reiter's default logic and some preliminaries. Section 4 presents our transformation and describes how tasks in default theory are mapped into equivalent tasks in propositional logic. Section 5 discusses the merits of acyclic theories, section 6 presents new procedures for query processing and identifies tractable classes using constraint networks techniques. Section 7 provides concluding remarks. Due to space considerations all proofs are omitted. For more details see [BED91].

2 Reiter's Default Logic

Following is a brief introduction to Reiter's default logic [Rei80]. Let $\mathcal{L}$ be a first order language. A ***default theory*** is a pair (D, W), where D is a set of defaults and W is a set of closed wffs (well formed formulas) in $\mathcal{L}$. A ***default*** is a rule of the form $\alpha : \beta_1, ..., \beta_n/\gamma$, where $\alpha, \beta_1, ...\beta_n$ and γ are formulas in $\mathcal{L}$. The intuition behind a default can be: If I believe in α and I have no reason to believe that one of the β_i is false, then I can believe γ. A default $\alpha : \beta/\gamma$ is *normal* if $\gamma = \beta$. A default is *semi-normal* if it is in the form $\alpha : \beta \wedge \gamma/\gamma$. A default theory is *closed* if all the first order formulas in D and W are closed.

The set of defaults D induces an *extension* on W. Intuitively, an extension is a maximal set of formulas that can be deduced from W using the defaults in D. Let $Th(E)$ denote the logical closure of E in $\mathcal{L}$. We use the following definition of an extension:

Definition 2.1 *([Rei80],theorem 2.1) Let $E \subseteq \mathcal{L}$ be a set of closed wffs, and let (D, W) be a closed default theory. Define*

- $E_0 = W$
- *For $i \geq 0$ $E_{i+1} = Th(E_i) \bigcup \{\gamma | \alpha : \beta_1, ..., \beta_n/\gamma \in D$ where $\alpha \in E_i$ and $\neg\beta_1, ...\neg\beta_n \notin E\}$*

E is an extension for (D, W) iff for some ordering $E = \bigcup_{i=0}^{\infty} E_i$. (Note the appearance of E in the formula for E_{i+1}).

Most tasks on a default theory (D, W) can be formulated using one of the following queries:

Existence : Does (D, W) have an extension ? If so, find one.

Set-Membership : Given a set of formulas S, Is S contained in some extension of (D, W)?

Set-Entailment : Given a set of formulas S, Is S contained in every extension of (D, W)?

In this paper we restrict our attention to Propositional, Disjunction-free, Semi-normal Default theories, denoted **PDSD**, (where formulas in D and W are disjunction-free). This is the same subclass studied by Kautz and Selman ([KS89]). We can assume, w.l.g., that W is consistent and that no default has a contradiction as a justification, since when W is inconsistent, only one extension exists and a rule having contradictory justification can be eliminated by inspection.

3 Definitions and Preliminaries

We denote propositional symbols by upper case letters $P, Q, R...$, propositional literals (i.e. $P, \neg P$) by lower case letters $p, q, r...$ and conjunctions of literals by $\alpha, \beta....$ The operator $\sim$ over literals is defined as follows: If $p = \neg Q$, $\sim p = Q$, If $p = Q$ then $\sim p = \neg Q$. If $\delta = \alpha : \beta/\gamma$ is a default, we define $\text{pre}(\delta) = \alpha$, $\text{just}(\delta) = \beta$ and $\text{concl}(\delta) = \gamma$.

We denote by S^* the *logical closure* of a set of formulas S and call S a *logical kernel* of S^*. Clearly, when dealing with PDSDs, every extension E^* has a logical kernel consisting of literals only. We assume, w.l.g. that the consequent in each rule is a single literal.

We say that a set of literals E *satisfies the preconditions of* δ if $\text{pre}(\delta) \in E$ and for each $q \in \text{just}(\delta)$ $\sim q \notin E$ [1]. We say that E *satisfies the rule* δ if it does not satisfy the preconditions of δ or else, it satisfies both its preconditions and its conclusion.

[1] Note that since we are dealing with PDSDs, if α is not a contradiction, the negation of one of its conjuncts is in the extension iff the negation of α is there too.

A *proof of a literal* p, w.r.t. a given set of literals E and a given PDSD (D, W) is a sequence of rules $\delta_1, ..., \delta_n$ such that the following three conditions hold : 1. $\text{concl}(\delta_n) = p$. 2. For all $1 \leq i \leq n$ and for each $q \in \text{just}(\delta_i)$, $\neg q \notin E$ 3. For all $1 \leq i \leq n$ $\text{pre}(\delta_i) \subseteq W \bigcup \{\text{concl}(\delta_1), ..., \text{concl}(\delta_{i-1})\}$.

The following lemma is instrumental throughout the paper. It can be viewed as the declarative counterpart of lemma 1 in [KS89].

Lemma 3.1 *E^* is an extension of a PDSD (D, W) iff E^* is a logical closure of a set of literals E that satisfies :*

1. $W \subseteq E$
2. *E satisfies each rule in D.*
3. *For each $p \in E$, there is a proof of p in E .* □

We define the *dependency graph* $G_{(D,W)}$, of a PDSD (D, W), to be a directed graph constructed as follows: Each literal p appearing in D or in W is associated with a node, and an edge is directed from p to r if there is a default rule where p appears in its prerequisite and r is its consequent and $r \notin W$. An *acyclic PDSD* is one whose dependency graph is acyclic, a property that can be tested linearly (see [Tar72]).

4 Expressing PDSD in propositional logic

Our aim is to transform a given a PDSD (D, W), to a set of propositional sentences $\mathcal{P}_{(D,W)}$ such that $\mathcal{P}_{(D,W)}$ has a model iff (D, W) has an extension, and vice-versa, every model of $\mathcal{P}_{(D,W)}$ has a corresponding extension for (D, W). Since the transformation of acyclic PDSDs is simpler, (both in terms of exposition and in terms of complexity), we present it separately, and later extend it to the cyclic case.

4.1 The acyclic Case

The reason acyclic PDSD are simpler is that their extensions can be expressed (and can also be generated) in a more relaxed fashion. This is demonstrated through Lemma 4.1, a relaxed version of the general Lemma 3.1. We can show that (note the change in item 3) :

Lemma 4.1 *E^* is an extension of an* acyclic *PDSD (D, W) iff E^* is a logical closure of a set of literals E that satisfies :*

1. $W \subseteq E$
2. *E satisfies each rule in D.*
3. *for each $p \in E - W$ there is $\delta \in D$ such that $\text{concl}(\delta) = p$ and E satisfies the preconditions of δ.* □

Expressing the above conditions in propositional logic, results in a propositional theory whose models coincide with the extensions of the acyclic default theory. Let $\mathcal{L}$ be the underlying propositional language of (D, W). For each propositional symbol in $\mathcal{L}$, we define

two propositional symbols, I_P and $I_{\neg P}$ yielding a new set of symbols : $\mathcal{L}' = \{I_P, I_{\neg P} | P \in \mathcal{L}\}$. Intuitively, I_P stands for "P is in the extension" while $I_{\neg P}$ stands for "$\neg P$ is in the extension".

To simplify notations we use the notions of $in(\alpha)$ and $cons(\alpha)$ that stand for "α is in the extension", and "α is consistent with the extension", respectively. Formally, $in(\alpha)$ and $cons(\alpha)$ are defined as functions from conjuncts in $\mathcal{L}$ to conjuncts in $\mathcal{L}'$ as follows :

- if $\alpha = P$ then $in(\alpha) = I_P$, $cons(\alpha) = \neg I_{\neg P}$.
- if $\alpha = \neg P$ then $in(\alpha) = I_{\neg P}$, $cons(\alpha) = \neg I_P$.
- if $\alpha = \beta \wedge \gamma$ then $in(\alpha) = [in(\beta)] \wedge [in(\gamma)]$, $cons(\alpha) = [cons(\beta)] \wedge [cons(\gamma)]$.

The following procedure,**translate-1**, translates an acyclic PDSD (D, W), to a set of sentences $\mathcal{P}_{(D,W)}$ in propositional logic as follows:

translate-1$((D, W))$

1. for each $p \in W$, put I_p into $\mathcal{P}_{(D,W)}$.
2. for each $\alpha : \beta/\gamma \in D$, if $\gamma \notin W$, add $in(\alpha) \wedge cons(\beta) \longrightarrow in(\gamma)$ into $\mathcal{P}_{(D,W)}$.
3. Let $S_p = \{[in(\alpha) \wedge cons(\beta)] | \exists \delta \in D$ such that $\delta = \alpha : \beta/p\}$.
   For each $p \notin W$, if $S_p \neq \emptyset$ then add to $\mathcal{P}_{(D,W)}$ the formula $I_p \longrightarrow [\vee_{\alpha \in S_p} \alpha]$.
   else, (If $p \notin W$ and $S_p = \emptyset$), add to $\mathcal{P}_{(D,W)}$ the formula $\neg I_p$. □

We claim that:

Theorem 4.2 *Procedure* translate-1 *transforms an acyclic PDSD* (D, W) *into propositional sentence,* $\mathcal{P}_{(D,W)}$, *such that* θ *is a model for* $\mathcal{P}_{(D,W)}$ *iff* $\{p | \theta(I_p) = \textbf{true}\}^*$ *is an extension for* (D, W). □

Algorithm translate-1 is time and space linear in $|D + W|$ (assuming W is sorted).

Example 4.3 ***(This example is based on Reiter's example 2.5)***

Consider the following acyclic PDSD : $D = \{A : P/P, : A/A, \neg A/\neg A\}$, $W = \emptyset$.

$\mathcal{P}_{(D,W)}$ = { (remains empty after step 1),

(following step 2:) $I_A \wedge \neg I_{\neg P} \longrightarrow I_P$, $\neg I_{\neg A} \longrightarrow I_A$, $\neg I_A \longrightarrow I_{\neg A}$,

(following step 3:) $I_P \longrightarrow I_A \wedge \neg I_{\neg P}$, $I_A \longrightarrow \neg I_{\neg A}$, $I_{\neg A} \longrightarrow \neg I_A$, $\neg I_{\neg P}\}$

$\mathcal{P}_{(D,W)}$ has only 2 models : $\{I_A = \textbf{true}, I_{\neg A} = \textbf{false}, I_{\neg P} = \textbf{false}, I_P = \textbf{true}\}$, that corresponds to the extension $\{A, P\}$, and $\{ I_A = \textbf{false}, I_{\neg A} = \textbf{true}, I_{\neg P} = \textbf{false}, I_P = \textbf{false}\}$, that corresponds to the extension $\{\neg A\}$. □

4.2 The Cyclic Case

Since procedure *translate-1* assumes acyclic PDSD, it was not careful to eliminate the possibilities of unfounded proofs. If applied to cyclic PDSD, the resulting transformation will possess models that corresponds to illegal extensions, i.e., ones that are generated by cyclic proofs [BED91]. Thus, an extended translation is needed.

The common approach for building an extension,(used by Etherington ([Eth87]), Kautz and Selman ([KS89]), and others), is to increment W using rules from D. By formulating the default theory as a set of constraints on the set of its extensions, we make a declarative account of such process, thus allowing all the general techniques of constraint satisfaction and propositional satisfiability to be used. This frees us from worrying about orderness. however it requires adding a constraint guaranteeing that if a formula is in the extension. there is a series of defaults deriving it from W.

In the acyclic case, this was achieved by the third constraint in Lemma 4.1 implemented by step 4 of procedure *translate-1*. However, in cyclic PDSDs we must add the constraint that if a literal, not in W, belongs to the extension, then the prerequisite of at least one of its rules should be in the extension *on its own rights*, namely, not as a consequence of a circular proof. One way to avoid circular proofs is to impose indexing on literals such that for every literal in the extension there exist a proof with literals having lower indices.

To implement this idea, originally mentioned at [Dis89], we associate an *index variable* with each literal in the transformed language, and require that, p is in the extension, only if it is the consequent of a rule whose prerequisite's indexes are smaller. Let $\#p$ stand for the "index associated with p", and let k be its number of values. These new multi-valued variables can be expressed in propositional logic using additional $O(k^2)$ clauses and literals (see [BED91]). For simplicity, however, we will use the multi-variable notations, viewing them as abbreviations to their propositional counterparts.

Let $\mathcal{L}''$ be the language $\mathcal{L}' \bigcup \{\#p | p \in \mathcal{L}\}$, where $\mathcal{L}'$ is the set $\{I_p, I_{\neg P} | P \in \mathcal{L}\}$ defined earlier. Procedure **translate-2** transforms any PDSD (cyclic or acyclic) over $\mathcal{L}$ to a set of propositional sentences over $\mathcal{L}''$as follows :

procedure translate-2(D, W)

1. for each $p \in W$ put I_p into $\mathcal{P}_{(D,W)}$.
2. for each $\alpha : \beta/\gamma \in D$, add $in(\alpha) \wedge cons(\beta) \longrightarrow in(\gamma)$ to $\mathcal{P}_{(D,W)}$.
3. Let $C_p = \{[in(q_1 \wedge q_2 ... \wedge q_n) \wedge cons(\beta)] \wedge [\#q_1 < \#p] \wedge ... \wedge [\#q_n < \#p] \mid \exists \delta \in D$ such that $\delta = q_1 \wedge q_2 ... \wedge q_n : \beta/p \}$.

   For each $p \notin W$, if C_p is not empty then, add to $\mathcal{P}_{(D,W)}$ the formula $I_p \longrightarrow [\vee_{\alpha \in C_p} \alpha]$.

   Else, (If $p \notin W$ and $C_p = \emptyset$) add $\neg I_p$ to $\mathcal{P}_{(D,W)}$.

The complexity of this translation requires adding n index variables, n being the number of literals in $\mathcal{L}$, each having at most n values. Since expressing an *inequality* in propositional logic, requires $O(n^2)$ clauses, and since there are at most n possible inequalities per default, the resulting size of this transformation is bounded by $O(|W| + |D|n^3)$ propositional sentences.

The following theorems summarize the properties of our transformation. In all of them, $\mathcal{P}_{(D,W)}$ is the set of sentences resulting from translating a given PDSD (D, W) using translate-2 (or translate-1 when the theory is acyclic).

Theorem 4.4 *Let (D, W) be a PDSD. If $\mathcal{P}_{(D,W)}$ is satisfiable and if θ is a model for $\mathcal{P}_{(D,W)}$, then $\{p | \theta(I_p) = \mathbf{true}\}^*$ is an extension for (D, W).* □

Theorem 4.5 *If E^* is an extension for (D,W) then there is a model θ for $\mathcal{P}_{(D,W)}$ such that $\theta(in(p))$ = **true** iff $p \in E^*$.* □

Corollary 4.6 *A PDSD (D,W) has an extension iff $\mathcal{P}_{(D,W)}$ is satisfiable.* □

Corollary 4.7 *A set of literals, S, is contained in an extension of (D,W) iff there is a model for $\mathcal{P}_{(D,W)}$ which satisfies the set $\{I_p | p \in S\}$.* □

Corollary 4.8 *A literal p is in every extension of a PDSD (D,W) iff there is no model for $\mathcal{P}_{(D,W)}$ which satisfies $\neg I_p$.* □

The above theorems suggest that we can first translate a given PDSD (D,W) to $\mathcal{P}_{(D,W)}$ and then answer queries as follows: to test if (D,W) has an extension, we test satisfiability of $\mathcal{P}_{(D,W)}$, to see if a set S of literals is a member in some extension, we test satisfiability of $\mathcal{P}_{(D,W)} \cup \{I_p | p \in S\}$ and to see if S is included in every extension, we test if for every $p \in S$, $\mathcal{P}_{(D,W)} \cup \neg I_p$ is not satisfiable.

4.3 An improved translation

A closer look at procedure translate-2 reveals that it can be further improved. If a prerequisite of a rule is not on a cycle with its consequent, we do not need to index them, nor to enforce the partial order among their variables. Thus, we need indexes only for literals which reside on cycles in the *dependency graph*. Furthermore, since we will never have to solve cyclicity between two literals that do not share a cycle, the range of the index variables is bounded by the maximum number of literals that share a common cycle. Actually , we show that the index variable's range can be bounded by the maximal length of an acyclic path in any *strongly connected component* in $G_{(D,W)}$(see [BED91]). The strongly-connected components of a directed graph is a partition of its set of nodes such that for each subset C in the partition, and for each $x, y \in C$, there is a directed path from x to y and from y to x in G. The strongly connected components can be identified in linear time [Tar72].

Procedure *translate-3* incorporates these observations by revising step 4 of translate-2. The procedure associates index variables only with literals that are part of a non-trivial cycle (i.e. cycle with at least two nodes).

procedure translate-3((D,W))-step 4

4.a Identify the strongly connected components of $G_{(D,W)}$.

4.b Let $C_p = \{[in(q_1 \wedge q_2 ... \wedge q_n) \wedge cons(\beta)] \wedge [\#q_1 < \#p] \wedge ... \wedge [\#q_r < \#p] \mid \exists \delta \in D$ such that $\delta = q_1 \wedge q_2 ... \wedge q_n : \beta / p$, and $q_1, ..., q_r$ $(r \leq n)$ are in p's component $\}$ (if p's components contains only p, $C_p = \emptyset$.)

Let $A_p = \{[in(\alpha) \wedge cons(\beta)] \mid$ Exists $\delta \in D$ such that $\delta = \alpha : \beta / p$ and no literal in α is in the same component as $p\}$.

Let $S_p = A_p \cup C_p$.

For each $p \notin W$ add $I_p \longrightarrow [\vee_{\alpha \in S_p} \alpha]$ to $\mathcal{P}_{(D,W)}$.

If $p \notin W$ and $S_p = \emptyset$ add $\neg I_p$ to $\mathcal{P}_{(D,W)}$. □

Procedure translate-3 will behave exactly as procedure translate-1 when the input is an acyclic PDSD. The number of index variables produced by translate-3, is bounded by $\text{Min}\{k*c, n\}$, where k is the size of a largest component of $G_{(D,W)}$, c is the number of non-trivial components and n the number of literals in the language. The range of the index variable is bounded by l – the length of the longest acyclic path in any component ($l \leq k$). Since in each rule we have at most k literals in the prerequisite that share a component with its consequence, the resulting propositional transformation is bounded by additional $O(|W| + |D|kl^2)$ sentences, giving an explicit connection between the complexity of the transformation and its cyclicity level, as reflected by k and l. Theorems 4.4 through 4.8 hold for procedure translate-3 as well.

5 Acyclicity and orderness

We saw that acyclic PDSDs allow a smoother transformation. We next present a nondeterministic algorithm for finding an extension of an acyclic PDSD. It is justified by lemma 4.1 and it is simpler than the algorithm presented by Kautz and Selman ([KS89]) for a *general* PDSD.

Acyclic-find extension input : an acyclic PDSD (D, W) .

1. Let $V = \{p|$ there is a rule in D with p as a consequent $\}$.
2. Guess $E \subseteq (V \bigcup W)$ such that E is a superset of W.
3. Check that conditions 2 and 3 of lemma 4.1 are satisfied by E. If so, E^* is an extension of (D, W). □

While we distinguish between cyclic and acyclic PDSDs, Etherington ([Eth87]) has distinguished between ordered and unordered default theories. He has defined an order induced by the defaults in D on the set of literals, and showed that if a semi-normal theory is ordered, than it has at least one extension.

To understand the relationship between these two categories we define a *generalized dependency graph* of a PDSD, to be a directed graph with blue and white arrows. Each literal is associated with a node in the graph, and for every $\delta = \alpha : \beta/p$ in D, every $q \in \alpha$, and every $r \in \beta$, there is a blue edge from q to p and a white edge from $\sim r$ to p. A PDSD is *unordered* iff its generalized dependency graph has a cycle having at least one white edge. A PDSD is *cyclic* iff its generalized dependency graph has a blue cycle (i.e. a cycle with no white edges). Note that a set of default rules which is ordered is not necessarily acyclic and vice versa. The following set of rules $\{P : Q/Q, Q : P/P\}$ is ordered but cyclic while the set $\{P : Q/Q, S : \neg Q \wedge P/P\}$ is acyclic but not ordered .

Clearly, the expressive power of the ordered and the acyclic subsets of PDSD is limited ([KS89]). Cyclic theories are needed, in particular, for characterizing two properties which are highly correlated. For example, to express the belief that usually people who smoke drink and vice versa, we need the defaults Drink : Smoke /Smoke, Smoke : Drink / Drink, yielding a cyclic default theory. The characterization of default theories presented in the following section may be viewed as a generalization of both acyclicity and orderness.

6 More tractable subsets for default logic

What do we gain from the above transformation in terms of the complexity of processing the default logic? Although we aimed at introducing topological considerations, we have looked at syntactical features as well.

It is easy to characterize the syntax of the propositional sentences generated by our transformation. Since this translation is time and space polynomial (in the size of the default theory), when its resulting output belongs to a tractable propositional subclass, all processing tasks of existence, set-membership and set-entailment, can be performed efficiently.

One such subset, already identified by [KS89] and [Sti90], is the "Prerequisite free normal unary" (a PDSD with normal rules having no prerequisite). This subset is translated to sentences in *2-SAT*, a subclass containing disjunctions of at most two literals, whose linear satisfiability ([EIS76]) induces a linear time algorithm for the default theory tasks. In contrast, Kautz and Selman presented a quadratic algorithm (for deciding "membership in all extensions") applicable to a broader class of PDSDs (called "normal unary") where the prerequisite of each (normal) rule consists of a single positive literal.

Since propositional satisfiability can be regarded as a constraint satisfaction problem, we use techniques borrowed from that field to solve satisfiability. We next outline the general prospects involved in using constraint networks techniques and will demonstrate their utilities. However, for a full account of this approach see [BED91].

In general, constraint satisfaction techniques exploit the structure of the problem through the notion of a "constraint graph". For propositional sentences, the constraint graph (also called a "primal constraint graph") associates a node with each propositional letter and connects any two nodes whose associated letters appear in the same propositional sentence. Various graph parameters were shown as crucially related to solving the satisfiability problem. These include the *induced width*, $,w^*$, the *size of the cycle-cutset*, the *depth of a depth-first-search spanning tree* of this graph and the *size of the non-separable components* ([Fre85]),[DP88]). It can be shown that the worse-case complexity of deciding consistency is polynomially bounded by any one of these parameters.

Since, these parameters can be bounded easily by simple processing of the given graph, they can be used for assessing tractability ahead of time. For instance, when the constraint graph is a tree, satisfiability can be answered in linear time. In the sequel we will demonstrate the potential of this approach using one specific technique, called **Tree-Clustering** [DP89], customized for solving propositional satisfiability, and emphasize its effectiveness for maintaining a default data-base.

The *Tree-Clustering* scheme has a *tree-building* phase, and a *query processing* phase. The complexity of the former is exponentially dependent on the sparseness of the constraint graph, while the complexity of the latter is always linear in the size of the data-base generated by the *tree-building* preprocessing phase. Consequently, even when building the tree is computationally expensive it may be justified when many queries on the same PDSD are expected. The algorithm is summarized below (for details see [DP89]).

Propositional-Tree-Clustering (tree-building)

input: a set of propositional sentences S and its constraint graph.

1. Use the *triangulation* algorithm to generate a *chordal* constraint graph.

A graph is *chordal* if every cycle of length at least four has a chord.

The ***triangulation algorithm*** [TY84] transforms any graph into a chordal graph by adding edges to it. It consists of two steps:

(a) Select an ordering for the nodes, (various heuristics for good orderings are available).

(b) Fill in edges recursively between any two nonadjacent nodes that are connected via nodes higher up in the ordering.

2. Identify all the *maximal cliques* in the graph. Let $C_1, ..., C_t$ be all such cliques indexed by the rank of their highest nodes.

3. Connect each C_i to an ancestor C_j $(j < i)$ with whom it shares the largest set of letters. The resulting graph is called a **join tree.**

4. Compute $\mathcal{M}_i$, the set of models over C_i that satisfy S_i, where S_i be the set of all sentences composed only of letters in C_i.

5. For **each** C_i and for *each* C_j adjacent to C_i in the join tree, delete from $\mathcal{M}_i$ every model M that has no model in $\mathcal{M}_j$ that agrees with it on the set of their common letters. This amounts to performing *arc consistency* on the join tree. □

Since the most costly operation within the ***tree-building*** algorithm is generating all the submodels of each clique (step 5), the time and space complexity of this preliminary phase is $O(n * 2^{|C|})$, where $|C|$ is the size of the largest clique and n is the number of letters used in S . It can be shown that $|C| = w^* + 1$, where w^* is the ***width*** [2] of the ordered chordal graph (also called *induced width* [DP89]. As a result, for problem classes having a *bounded induced width*, this method is tractable.

Once the tree is built it always allows an efficient query processing. This procedure is described within the following general scenario. (n stands for the number of letters in the original PDSD, m, bounds the number of submodels for each clique.) [3]

1. Translate the PDSD to propositional logic (generates $O(|W| + |D|n^3)$ sentences)

2. Build a default data-base from the propositional sentences using the *Tree-building* method (takes $O(n^2 * \exp(w^* + 1))$).

3. Answer queries on the default theory using the produced tree:

   - To **answer** whether there is an extension, test if there is an empty clique. If so, no **extension** exists. (bounded by $O(n^2)$ steps).
   - To **find an extension**, solve the tree in a backtrack-free manner:
     In order to find a satisfying model we pick an arbitrary node C_i in the join tree, select a model M_i from $\mathcal{M}_i$, select, from each of its neighbors C_j, a model M_j

[2] The ***width*** of a node in an ordered graph is the number of edges connecting it to nodes lower in the ordering. The width of an ordering is the maximum width of nodes in that ordering, and the width of a graph is the minimal width of all its orderings

[3] Note that the number of letters in the propositional sentences is $O(n^2)$ if the PDSD is cyclic, and $O(n)$ if it is acyclic, and that m is not larger than the total number of extensions.

that agrees with M_i on common letters, unite all these models and continue to the neighbors' neighbors, and so on. The set of all models can be generated by exhausting all combinations of submodels that agree on their common letters. (finding one model is bounded by $O(n^2 * m)$ steps)

- To answer whether there is an extension that satisfy a set of literals A, check if there is a model satisfying $\{I_p | p \in A\}$. (This takes $O(n^2 * m * logm)$ steps).
- To answer whether a literal p is included in all the extensions, check whether there is a solution that satisfies $\neg I_p$, (bounded by $O(n^2 m)$ steps).

Note that we could translate the PDSD directly to a set of constraints and then use the above techniques, thus getting less variables by representing the index variables as multi-valued variable. However, since the first part of the paper was devoted to show a translation from PDSD to propositional logic, we chose to continue with this language.

Following is an example demonstrating our approach.

Example 6.1 Consider the following PDSD :

$$D = \{\frac{\text{Dumbo : Elephant} \wedge \text{Fly}}{\text{Elephant}} \quad \frac{\text{Elephant} \wedge \neg\text{Fly} : \neg\text{Dumbo}}{\neg\text{Dumbo}}$$
$$\frac{\text{Elephant : } \neg\text{Fly}}{\neg\text{Fly}} \quad \frac{\text{Dumbo : Fly}}{\text{Fly}}$$
$$\frac{\text{Elephant : } \neg\text{Circus}}{\neg\text{Circus}} \quad \frac{\text{Dumbo : Elephant} \wedge \text{Circus}}{\text{Circus}}\}$$
$$W = \{\text{Dumbo, Elephant}\}$$

The propositional letter "Dumbo" represents here a special kind of elephants that can fly. These defaults state that normally, Dumbos, assuming they fly, are elephants, if an elephant does not fly we do not believe that it is a Dumbo. Elephants usually do not fly, while Dumbos usually fly. Most elephants are not living in a circus while Dumbos usually live in a circus.

This is an acyclic default theory, thus algorithm *translate-1* when applied produces the following set of sentences (each proposition is abbreviated by its initial letter):

Sentences generated in step 2 of translate-1: I_D, I_E.

step 3 :

$I_E \wedge I_{\neg F} \wedge \neg I_D \longrightarrow I_{\neg D}$, $I_E \wedge \neg I_F \longrightarrow I_{\neg F}$, $I_D \wedge \neg I_{\neg F} \longrightarrow I_F$, $I_E \wedge \neg I_C \longrightarrow I_{\neg C}$,
$I_D \wedge \neg I_{\neg E} \wedge \neg I_{\neg C} \longrightarrow I_C$.

step 4 :

$I_{\neg D} \longrightarrow I_E \wedge I_{\neg F} \wedge \neg I_D$, $I_{\neg F} \longrightarrow I_E \wedge \neg I_F$, $I_F \longrightarrow I_D \wedge \neg I_{\neg F}$, $I_{\neg C} \longrightarrow I_E \wedge \neg I_C$,
$I_C \longrightarrow I_D \wedge \neg I_{\neg E} \wedge \neg I_{\neg C}$, $\neg I_{\neg E}$

The primal **graph** of this set is shown in figure 1. It is already chordal and the ordering $I_E, I_{\neg F}, I_D, I_{\neg D}, I_{\neg C}, I_C, I_F, I_{\neg E}$ suggests that for this particular problem, $w^* \leq 3$. Thus, using the *tree-Clustering* method we can answer queries about extension, set-membership and set-entailment in polynomial time (bounded by exp(4)). Note that this PDSD is unordered and not unary, therefore, the complexity of answering queries for such PDSD is NP-hard [KS89].

We conclude this section with a characterization of the tractability of DSPD theories as a function of the *induced width*, w^*, of their *interaction graph*. The interaction graph is an undirected graph, where each literal in W or D is associated with a node and, for every

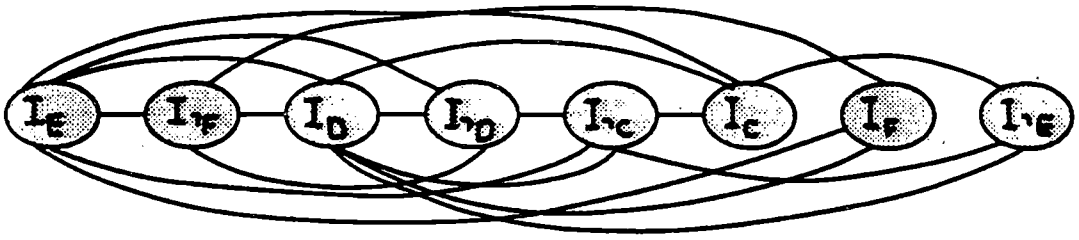


Figure 1: Constraints graph for example 6.1

$\delta = \alpha : \beta / p$ in D, every $q \in \alpha$ and every $\sim r$ such that $r \in \beta$, there are arcs connecting all of them into one clique with p. (this graph can be extracted from the generalized dependency graph, by connecting all the parents of every node, disregarding colors and directionality).

Theorem 6.2 *A PDSD (D, W) whose interaction graph has an induced width w^* can decide existence, membership and entailment in $O(2^{w^*+1})$ when the theory is acyclic and $O(n^{w^*+1})$ when the theory is cyclic.* □

7 Summary and Conclusions

This paper presents a transformation of a disjunction-free semi-normal default theory into sentences in propositional logic such that the set of models of the latter coincides with the set of extensions of the former. Questions of existence, membership and entailment posed on the default theory are thus transformed into equivalent satisfiability problems. This mapping can be further formulated as consistency of constraint networks.

These mappings are valuable as they bring problems in non-monotonic reasoning into the familiar arenas of propositional satisfiability and constraint satisfaction problems. Previously, computational issues of non-monotonic theories were addressed by mapping them into *truth-maintenance systems (TMS)* or *ATMS* ([RDB89], [JK90], [Elk90],[dK86]). Our tansformation, take us one step further, since propositional logic and constraint networks accumulated a large body of theoretical understanding.

Using our transformation, we showed that default theories whose *interaction graph* has a bounded $w*$ are tractable, and can be solved in time and space bounded by $O(n^{w*+1})$ steps. This permits us to predict worse-case performance prior to processing, since w^* can be bounded in time quadratic in the number of literals. Moreover, the *tree-clustering* procedure, associated with the w^* analysis, provides an effective preprocessing strategy for maintaining the knowledge; once applied, all incoming queries can be answered swiftly and changes to the knowledge can often be incorporated in linear time.

In the full paper we show how additional tractable classes can be identified using other CSP techniques , including cycle-cutset , non-separable component, backjumping and others [BED91]. We conjecture that our transformation can be carried over to default theories having disjunctive sentences, thus permitting topological characterization of disjunctive semi-normal default theories as well.

References

[BED91] Rachel Ben-Eliyahu and Rina Dechter. Expressing default theories in constarint

language. Technical report, UCLA, 1991. in preparation.

[BF88] N. Bidiot and C. Froidevaux. General logical databases and programs : Default logic semantics and stratification. *Journal of Information and Computation*, 1988.

[Dis89] 1989. Paul Morris suggested it in an e-mail discussion following the constraints processing workshop in AAAI-89 , on the question of representing a TMS in constraints language.

[dK86] Johan de Kleer. Extending the atms. *Artificial Intelligence*, 28:163–196, 1986.

[dK89] Johan de Kleer. A comparison of atms and csp techniques. In *11th International Joint Conference on Artificial Intelligence*, pages 290–296, Detroit,Michigan, USA, 1989.

[DP88] Rina Dechter and Judea Pearl. Network-based heuristics for constraint satisfaction problems. *Artificial Intelligence*, 34:1–38, 1988.

[DP89] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38:353–366, 1989.

[EIS76] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM journal of Computing*, 5(4):691–703, 1976.

[Elk90] Charles Elkans. A rational reconstruction of nonmonotonic truth maintenance systems. *Artificial Intelligence*, 43:219–234, 1990.

[Eth87] David W. Etherington. Formalizing nonmonotonic reasoning systems. *Artificial Intelligence*, 31:41–85, 1987.

[Fre82] E.C. Freuder. A sufficient condition for backtrack-free search. *J. ACM*, 29(1):24–32, 1982.

[Fre85] E.C. Freuder. A sufficient condition for backtrack-bounded search. *J. ACM*, 32(4):755–761, 1985.

[JK90] Ulrich Junker and Kurt Konolige. Computing the extensions of autoepistemic and default logics with a tms. In *The 8th national conference on AI*, pages 278–283, Boston,MA, 1990.

[Kon88] Kurt Konolige. On the relation between default and autoepistemic logic. *Artificial Intelligence*, 35:343–382, 1988.

[KS89] Henry A. Kautz and Bart Selman. Hard problems for simple default logics. In *The first international conference on principles of knowledge representation and reasoning*, pages 189–197, Toronto,Ontario,Canada, 1989.

[MF84] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1984.

[RDB89] Michael Reinfrank, Oskar Dressler, and Gerd Brewka. On the relation between truth maintenance and autoepistemic logic. In *11th International Joint Conference on Artificial Intelligence*, pages 1206–1212, Detroit,Michigan, USA, 1989.

[Rei80] Ray Reiter. A logic for default reasoning. *Artificial Intelligence*, 13:81–132, 1980.

[Sti90] Jonathan Stillman. It's not my default : The complexity of membership problems in restricted propositional default logics. In *The 8th national conference on AI*, pages 571–578, Boston,MA, 1990.

[Tar72] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal of Computing*, 1(2), June 1972.

[TY84] Robert E. Tarjan and M Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs and selectively reduce acyclic hypergraphs. *SIAM journal of Computing*, 13(3):566–579, 1984.

Satisfiability and Reasoning with Inconsistent Beliefs using Energy Minimization

Gadi Pinkas
Department of Computer Science
Washington University
pinkas@cics.wustl.edu

Abstract

Connectionist networks with symmetric weights (like Hopfield networks and Boltzman Machines) use gradient descent to find a minimum for quadratic energy functions. They can be seen as performing constraint satisfaction when soft or hard constraints are encoded in the energy function. We show an equivalence between the problem of satisfiability in propositional calculus and the problem of minimizing those energy functions. The equivalence is in the sense that for any satisfiable Well Formed Formula (WFF) we can find a quadratic function that describes it, such that the set of solutions that minimize the function is equal to the set of truth assignments that satisfy the WFF. We also show that in the same sense every quadratic energy function describes some satisfiable WFF. Algorithms are given to transform any propositional WFF into an energy function that describes it and vice versa.
High-order models that use Sigma-Pi units are shown to be equivalent to the standard quadratic models with additional hidden units. Algorithms are given to convert high-order networks to low-order ones and vice versa.
We extend propositional calculus by augmenting beliefs with penalties (positive real numbers). The extended calculus is useful in expressing default knowledge, preference between arguments, and reliability of assumptions coming from unreliable redundant source(s) of knowledge.
We show a proof theory and semantics for reasoning from such inconsistent set of beliefs. The proof procedure ($\vdash$) is based on entailment from *all* preferred maximal consistent subsets of beliefs, while the semantics ($\models$) is based on satisfaction by all preferred models (soundness and completeness is shown). We give an algorithm to translate any inconsistent set of propositional beliefs into a network that searches for a preferred model of the set. Another algorithm is given that translates any network to a set of penalized beliefs whose preferred models the net is searching. Finally we sketch a connectionist inference engine for the above theory.

1. Introduction

Finding **minima** for quadratic functions is the essence of symmetric connectionist models used for constraint **satisfaction** [Hopfield 82] [Hinton, Sejnowski 86] [Hinton89]. They are characterized by a recurrent **network** architecture, a symmetric weight matrix (with zero diagonal) and a quadratic energy function **that** should be minimized. Each unit asynchronously computes the gradient of the function and adjusts its activation value, so that energy decreases monotonically. The network eventually reaches equilibrium, settling on either a local or a global minimum. [Hopfield,Tank85] demonstrated that certain complex optimization problems can be stated as constraints expressed in quadratic energy functions and be approximated by these kind of networks.

There is a direct mapping between these models and quadratic energy functions. Every quadratic energy function can be translated into a corresponding network and vice versa. Most of the time we will not distinguish between the function and the network that minimizes it.

In this paper we first show an equivalence between the satisfiability search problem and the problem of connectionist energy minimization. For every WFF we can find a quadratic energy function such that

the values of the variables of the function at the minimum can be translated into a truth assignment that satisfies the original WFF and vice versa. Also, any quadratic energy minimization problem may be described as a satisfiable WFF that is satisfied for the same assignments that minimize the function. More details and formal proofs can be found in [Pinkas 90a] and [Pinkas 90b].

We then show that any set of propositional constraints (possibly augmented with weights) can be translated into a quadratic energy function whose minima correspond to preferred models of the set. We shall demonstrate that symmetric networks are natural platforms for propositional defeasible reasoning and for noisy knowledge bases. In fact we shall show that every such network can be seen as encapsulating a body of knowledge and as performing a search for a satisfying model of that knowledge.

Finally we sketch a connectionist inference engine capable of reasoning from incomplete and inconsistent knowledge.

2. Satisfiability and models of propositional formulas

A WFF is an expression that combines atomic propositions (variables) and connectives $(\vee, \wedge, \neg, \rightarrow, (,))$. A model (truth assignment) is a vector of binary values that assigns 1 ("true") or 0 ("false") to each of the variables. A WFF φ is satisfied by a model $\bar{x}$ if its characteristic function H_φ evaluates to "one" given the vector $\bar{x}$.

The characteristic function is defined to be $H_\varphi : 2^n \rightarrow \{0,1\}$ such that:

- $H_{x_i}(x_1,\ldots,x_n) = x_i$
- $H_{(\neg\varphi)}(x_1,\ldots,x_n) = 1 - H_\varphi(x_1,\ldots,x_n)$
- $H_{(\varphi_1 \vee \varphi_2)}(x_1,\ldots,x_n) = H_{\varphi_1}(x_1,\ldots,x_n) + H_{\varphi_2}(x_1,\ldots,x_n) - H_{\varphi_1}(x_1,\ldots,x_n) \times H_{\varphi_2}(x_1,\ldots,x_n)$
- $H_{(\varphi_1 \wedge \varphi_2)}(x_1,\ldots,x_n) = H_{\varphi_1}(x_1,\ldots,x_n) \times H_{\varphi_2}(x_1,\ldots,x_n)$
- $H_{(\varphi_1 \rightarrow \varphi_2)}(x_1,\ldots,x_n) = H_{(\neg\varphi_1 \vee \varphi_2)}(x_1,\ldots,x_n)$

The satisfiability search problem for a WFF φ is to find an $\bar{x}$ (if one exists) such that $H_{\varphi(\bar{x})} = 1$.

3. Equivalence between WFFs

We call the atomic propositions that are of interest for a certain application "visible variables" (denoted by $\bar{x}$). We can add additional atomic propositions called "hidden variables" (denoted by $\bar{t}$) without changing the set of relevant models that satisfy the WFF. The set of models that satisfy φ projected onto the visible variables is then called "the visible satisfying models" $(\{\bar{x} \mid (\exists \bar{t}) H_\varphi(\bar{x},\bar{t}) = 1\})$.

Two WFFs are equivalent if the set of visible satisfying models of one is equal to the set of visible satisfying models of the other.

A WFF φ is in Conjunction of Triples Form (CTF) if $\varphi = \bigwedge_{i=1}^{m} \varphi_i$ and every φ_i is a sub-formula of at most three variables.[1]

Every WFF can be converted into an equivalent WFF in CTF by adding hidden variables. Intuitively, we generate a new hidden variable for every binary connective (eg: $\vee, \rightarrow$) except for the top most one, and we "name" the binary logical operation with a new hidden variable using the connective $(\leftrightarrow)$.

[1] CTF differs from the familiar Conjunctive Normal Form (CNF). The φ_i's are WFFs of up to 3 variables that may include any logical connective and are not necessarily a disjunction of literals as in CNF. To put a bidirectional CTF clause into a CNF we would have to generate two clauses, thus $(A \leftrightarrow B)$ becomes $(\neg A \vee B) \wedge (A \vee \neg B)$.

EXAMPLE **3.1** Converting $\varphi = ((\neg((\neg A) \wedge B)) \rightarrow ((\neg C) \rightarrow D))$ into CTF:
From $(\neg((\neg A) \wedge B))$ we generate: $((\neg((\neg A) \wedge B)) \leftrightarrow T_1)$ by adding a new hidden variable T_1,
from $((\neg C) \rightarrow D)$ we generate: $(((\neg C) \rightarrow D) \leftrightarrow T_2)$ by adding a new hidden variable T_2,
for the top most connective $(\rightarrow)$ we generate: $(T_1 \rightarrow T_2)$.
The conjunction of these sub-formulas is :
$((\neg((\neg A) \wedge B)) \leftrightarrow T_1) \wedge (((\neg C) \rightarrow D) \leftrightarrow T_2) \wedge (T_1 \rightarrow T_2)$. It is in CTF and is equivalent to φ.

4. Energy functions

A k-order energy function is a function $E : \{0,1\}^n \rightarrow \mathcal{R}$ that can be expressed in a sum of products form with product terms of up to k variables: $E^k(x_1, \ldots, x_n) =$

$$\sum_{1 \le i_1 < i_2 < \cdots < i_k \le n} w_{i_1,\ldots,i_k} x_{i_1} \cdots x_{i_k} + \sum_{1 \le i_1 < \cdots < i_{k-1} \le n} w_{i_1,\ldots,i_{k-1}} x_{i_1} \cdots x_{i_{k-1}} + \cdots + \sum_{1 \le i \le n}^{1} w_i x_i + w$$

Quadratic energy functions are special cases:

$$\sum_{1 \le i < j \le n} w_{ij} x_i x_j + \sum_{i \le n} w_i x_i + w.$$

We can arbitrarily divide the variables of an energy function into two sets: visible variables and hidden variables.

We call the set of minimizing vectors projected onto the visible variables, "The visible solutions" of the minimization problem. $(\mu(E) = \{\bar{x} \mid (\exists \bar{t}) E(\bar{x}, \bar{t}) = min_{\bar{y},\bar{z}}\{E(\bar{y}, \bar{z})\}\})$.

We can always translate back and forth [Hopfield 82] between a quadratic energy function and a network with symmetric weights that minimize it (see figure 1). Further, we can use high-order networks [Sejnowski 86] to minimize high-order energy functions. In the high-order model each node is assigned a Sigma-Pi unit that updates its activation value using:

$$a_i = F\Big(\sum_{i_1 \cdots i \cdots i_k} -w_{i_1,\ldots,i,\ldots,i_k} \prod_{1 \le j \le k, i_j \ne i} x_{i_j} \Big)$$

Like in the quadratic case, there is a translation back and forth between k-order energy functions and symmetric high-order models with k-order Sigma-Pi units (see figure 2).

For every energy function $E(\vec{x}, \vec{t})$ with $\vec{t}$ hidden variables, we define $Erank_E(\vec{x}) = min_{\vec{y}}\{\vec{x}, \vec{y}\}$.
The $Erank_E$ function defines the energy value of all visible states (when the visible units are clamped and the hidden units are free to settle to a minimum), and in this sense it strongly characterizes the network's behavior.

5. The equivalence between high-order models and low-order models

We call two energy functions *strongly equivalent*, if their corresponding *Erank* functions are equal up to a constant difference; i.e: $Erank_{E_1} = Erank_{E_2} + c$.

Any high-order energy function can be converted into a strongly equivalent low-order one with additional hidden variables. In addition, any energy function with hidden variables can be converted into a strongly equivalent, (possibly) higher one by eliminating some or all of the hidden variables. These algorithms allow us to trade the computational power of Sigma-Pi units for additional simple units and vice versa.

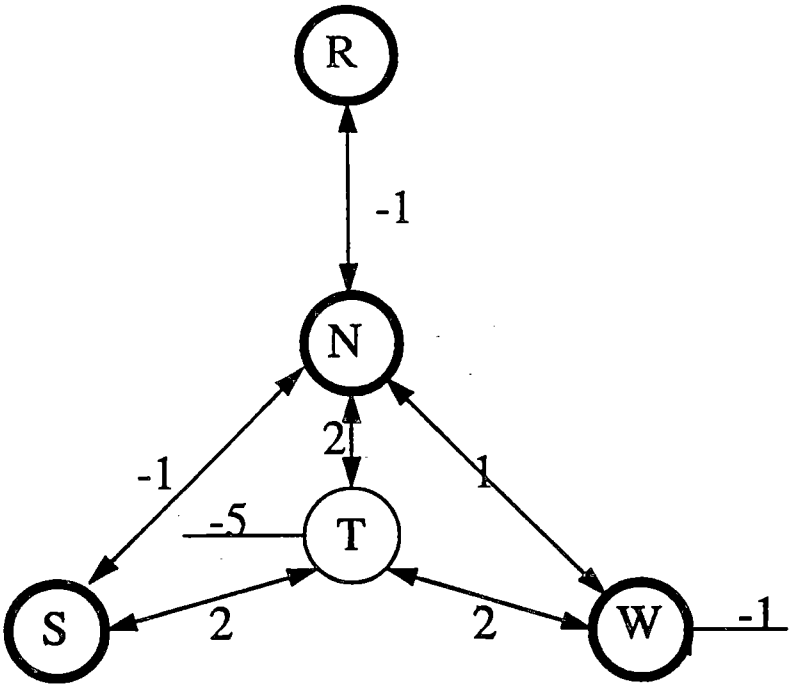


Figure 1: A symmetric network that represents the function $E = -2NT - 2ST - 2WT + 5T + NS + RN - WN + W$, describing the WFF: $(N \wedge S \rightarrow W) \wedge (R \rightarrow (\neg N)) \wedge (N \vee (\neg W))$. T is a hidden unit.

Theorem 1

- *Any k-order term* $(w \prod_{i=1}^{k} x_i)$*, with* NEGATIVE *coefficient* w*, can be replaced by the quadratic terms:* $\sum_{i=1}^{k} 2wX_iT - (2k-1)wT$ *generating a strongly equivalent energy function with one additional hidden variable* T.

- *Any k-order term* $(w \prod_{i=1}^{k} x_i)$*, with* POSITIVE *coefficient* w*, can be replaced by the terms:* $w \prod_{i=1}^{k-1} x_i - (\sum_{i=1}^{k-1} 2wX_iT) + 2wX_kT + (2k-3)wT$*, generating a strongly equivalent energy function of order* $k-1$ *with one additional hidden variable* T.

EXAMPLE 5.1 The cubic function $E = -NSW + NS + RN - WN + W$ is strongly equivalent to $-2NT - 2ST - 2WT + 5T + NS + RN - WN + W$, (introducing T). The corresponding high-order network appears in figure 2 while the equivalent quadratic one in figure 1.

The symmetric transformation, from low-order into high-order functions by eliminating any subset of the variables, is also possible (of course we are interesting in eliminating only hidden variables). To eliminate T, bring the energy function to the form: $E = E' + oldterm$, where $oldterm = (\sum_{j=1}^{k} w_j \prod_{i=1}^{l_j} X_{j_i})T$.
Consider all assignments S for the variables ($\hat{X} = x_{i_1} \cdots x_{i_l}$) in *oldterm* (not including T), such that $\beta_S = \sum_{j=1}^{k} w_j \prod_{i=1}^{l_j} x_{j_i} < 0$.
Each negative β_S represents an energy state of the variables in $\hat{X}$ that pushes T to become "one" and decreases the total energy by $\mid \beta_S \mid$. States with positive β_S cause T to become zero, do not reduce the total energy, and therefore can be ignored. Therefore, the only states that matter are those that reduce the energy; i.e β_S is negative.
Let $L_S^j = \begin{cases} \text{"}X_{i_j}\text{"} & \text{if } S(X_{i_j}) = 1 \\ \text{"}(1 - X_{i_j})\text{"} & \text{if } S(X_{i_j}) = 0 \end{cases}$ it is the expression "X_i" or "$(1 - X_i)$" depending whether

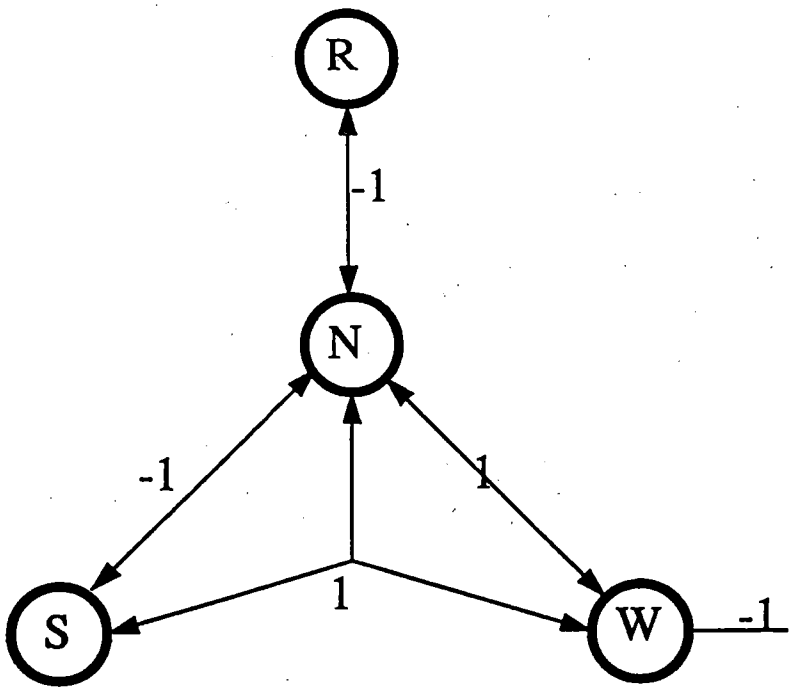


Figure 2: A cubic network that represents $E = -NSW + NS + RN - WN + W$ using Sigma-Pi units and a cubic hyper-arc. (Its is equivalent to the network of figure 1 without hidden units)

the variable is assigned 1 or 0 in S.
The expression $\prod_{j=1}^{l} L_S^j$ therefore determines the state S, and the expression

$$newterm = \sum_{\text{S such that } \beta_S < 0} \beta_S \prod_{j=1}^{l} L_S^j$$

represents the disjunction of all the states that cause a reduction in the total energy.
The new function $E' + newterm$, is therefore equivalent to $E' + oldterm$ and does not include T.

With this technique, we can convert any network with hidden units into a strongly equivalent network without any such units.

6. Describing WFFs using penalty functions

An energy function E describes a WFF φ if the set of visible satisfying models of φ is equal to the set of visible solutions of the minimization of E.

The penalty function E_φ of a WFF φ is a function $E_\varphi : \{0,1\}^n \rightarrow \mathcal{N}$, that penalizes sub-formulas of the WFF that are not satisfied. It computes the characteristic of the negation of every sub-formula φ_i in the upper level of the WFF's conjunctive structure.
If $\varphi = \bigwedge_{i=1}^{m} \varphi_i$ then,

$$E_\varphi = \sum_{i=1}^{m} (H_{\neg\varphi_i}) = \sum_{i=1}^{m} (1 - H_{\varphi_i})$$

If all the sub-formulas are satisfied, E_φ gets the value zero; otherwise, the function computes how many are unsatisfied.

It is easy to see that φ is satisfied by $\bar{x}$ iff E_φ is minimized by $\bar{x}$ (the global minima have a value of zero). Therefore, every satisfiable WFF φ has a function E_φ such that E_φ describes φ.

EXAMPLE 6.1

$$\begin{aligned} E_{((N\wedge S)\rightarrow W)\wedge(R\rightarrow(\neg N))\wedge(N\vee(\neg W))} &= H_{\neg((N\wedge S)\rightarrow W)} + H_{\neg(R\rightarrow(\neg N))} + H_{\neg(N\vee(\neg W))} \\ &= H_{N\wedge S\wedge(\neg W)} + H_{R\wedge N} + H_{(\neg N)\wedge W} \\ &= (NS(1-W)) + (RN) + ((1-N)W) \\ &= -NSW + NS + RN - WN + W \end{aligned}$$

The corresponding network appears in figure 2.

Theorem 2 *Every WFF is described by some quadratic energy function.*

The following algorithm transforms a WFF into a quadratic energy function that describes it, generating $O(length(\varphi))$ hidden variables:

- Convert into CTF (section 3).
- Convert CTF into a cubic energy function and simplify it to a sum of products form (section 6).
- Convert cubic terms into quadratic terms. Each of the triples generates only one new variable. (section 5).

The algorithm generates a network whose size is linear in the number of binary connectives of the original WFF. The fan-out of the hidden units is bounded by a constant.

7. Every energy function describes some satisfiable WFF.

In section 5 we saw that we can convert any energy function to contain no hidden variables. We show now that for any such function E with no hidden variables there exist a satisfiable WFF φ such that E describes φ.

The procedure is first to find the set $\mu(E)$ of minimum energy states (the vectors that minimize E). For each such state create an n-way conjunctive formula of the variables or their negations depending whether the variable is assigned 1 or 0 in that state. Each such conjunction $\bigwedge_{i=1}^{n} L_S^i$ where $L_S^i = \begin{cases} \text{“}X_i\text{”} & \text{if } S(X_i)=1 \\ \text{“}(\neg X_i)\text{”} & \text{if } S(X_i)=0 \end{cases}$ represents a minimum energy state. Finally the WFF is constructed by taking the disjunction of all the conjunctions: $\varphi = \bigvee_{S\epsilon\mu(E)}(\bigwedge_{i=1}^{n} L_S^i)$. The satisfying truth assignments of φ correspond directly to the energy states of the net.

We therefore conclude:

Theorem 3 *Every energy function describes some WFF.*

8. Reasoning from inconsistency and penalty calculus

We now extend propositional calculus by augmenting assumptions with penalties (like in [Derthick 88]). The extended calculus is able to deal with an inconsistent knowledge base (due to noise, errors in observations, unreliable sources, *etc...*) and can also be used as a framework for defeasible reasoning.

A ***Penalty Logic WFF*** (PLOFF) ψ is a finite set of pairs. Each pair is composed of a real positive number, called ***penalty***, and a standard propositional WFF, called ***assumption*** ; i.e., $\psi = \{< \rho_i, \varphi_i > | \ \rho_i \in \mathcal{R}^+, \varphi_i \text{ is a WFF}, i = 1...n\}$.

8.1. A proof-theory and semantics for penalty calculus

8.1.1. Proof theory. T is a *sub-theory* of a PLOFF ψ if T is a consistent subset (in the classical sense) of the assumptions in ψ; i.e., $T \subseteq \{\varphi_i \mid < \rho_i, \varphi_i > \in \psi\} = \mathcal{U}_\psi$, (note that $\mathcal{U}_\psi$ may be inconsistent).

The *penalty* of a sub-theory T of ψ is the sum of the penalties of the assumptions in ψ that are not included in T; i.e., $penalty_\psi(T) = \sum_{\varphi_i \in (\mathcal{U}_\psi - T)} \rho_i$ and is called the *penalty function* of ψ.

A *Minimum Penalty* sub-theory (MP-theory) of ψ is a sub-theory T that minimizes the penalty function of ψ; i.e., $penalty_\psi(T) = MIN_S\{penalty_\psi(S) \mid S \text{ is a sub-theory of } \psi\}$.

Let $T_\psi = \{T_i\}$ the set of all MP-theories of ψ, and let $T_\varphi = \{T'_j\}$ the set of all MP-theories of φ.
We say the ψ *entails* φ ($\psi \vdash \varphi$) iff all MP-theories of ψ entail (classical sense) the disjunction of all MP-theories of φ; i.e: $\bigvee T_i \vdash \bigvee T'_j$.
Note that when φ contains a consistent set of beliefs then, $\psi \vdash \varphi$ iff all MP-theories of ψ entail φ.

8.1.2. Model theory:. The *violation-rank* of a PLOFF ψ is the function ($Vrank_\psi$) that assigns a real-valued rank to each of the truth assignments. The $Vrank_\psi$ function is computed by summing the penalties for the assumptions of ψ that are violated by the assignment; i.e., $Vrank_\psi(\vec{x}) = \sum_i \rho_i H_{\neg\varphi_i}(\vec{x})$. The models that minimize the function are called *satisfying models* or preferred models.

We say that ψ *semantically entails* φ ($\psi \models \varphi$) iff the satisfying models of ψ also satisfy φ; i.e: $\mu(Vrank_\psi) \subseteq \mu(Vrank_\varphi)$ where μ is the set of minimizing vectors.

Theorem 4 *The proof procedure is sound and complete; i.e., $\psi \models \varphi$ iff $\psi \vdash \varphi$.*

8.2. Penalty calculus and energy functions

We say that a PLOFF ψ is strongly equivalent to an energy function E iff $(\forall \vec{x}) Vrank_\psi(\vec{x}) = Erank_E(\vec{x}) + c$.

Theorem 5 *For every PLOFF $\psi = \{< \rho_i, \varphi_i > \mid i = 1 \ldots n\}$ there exists a strongly equivalent quadratic energy function $E(\vec{x}, \vec{t})$.*

We can construct E from ψ using the following procedure:

1. "Name" all φ_i's using new hidden atomic propositions T_i and construct the set $\{< \infty, T_i \leftrightarrow \varphi_i >\}$. The high penalty guarantees that these WFFs will always be satisfiable.
2. Construct $\psi' = \{< \infty, T_i \leftrightarrow \varphi_i >\} \cup \{< \rho_i, T_i >\}$ so that the T_i's compete with each other.
3. Construct the energy function $\sum_i \beta E_{T_i \leftrightarrow \varphi_i} - \sum_j \rho_j T_j$, where β is chosen to be sufficiently large (practically ∞), and E_φ is the function generated by the algorithm of section 6.

The network that is generated can be seen as performing a search for a satisfying model of ψ. It can also be seen as searching for a MP-theory of ψ (at global minimum, the T_i's are activated for the φ_i's that are included in the MP-theory found).

Theorem 6 *Every energy function E is strongly equivalent to some PLOFF ψ;*

The following algorithm generates a strongly equivalent PLOFF from an energy function:

1. Eliminate hidden variables (if any) from the energy function using the algorithm of section 5.
2. The energy function (with no hidden variables) is now brought into a sum-of-products form and is converted into a PLOFF in the following way:
   Let $E(\vec{x}) = \sum_{i=1}^{m} w_i \prod_{n=1}^{k_i} x_{i_n}$ be the energy function.
   We construct a PLOFF $\psi = \{< -w_i, \bigwedge_{n=1}^{k_i} x_{i_n} > | \; w_i < 0\} \cup \{< w_l, \neg \bigwedge_{n=1}^{k_l} x_{l_n} > | \; w_l > 0\}$.

9. A sketch of a connectionist inference engine

Let ψ and φ be PLOFFs. We would like to construct a connectionist network to answer one of the possible three answers: 1) $\psi \models \varphi$; 2) $\psi \models (\neg\varphi)$; or 3) both $\psi \not\models \varphi$ and $\psi \not\models (\neg\varphi)$ ("unknown"). For simplicity let us first assume that φ is an atomic proposition. Later we'll describe a general solution.

Intuitively, our connectionist engine is built out of two sub-networks, each that is trying to find a satisfying model for ψ. The first sub-network is biased to search for a model which satisfies also φ, whereas the second sub-network is biased to search for a model which satisfies $\neg\varphi$. If two such models exist then we conclude that φ is "unknown". If no model of ψ also satisfies φ, we conclude that $\psi \models \neg\varphi$, and if no model of ψ satisfies $\neg\varphi$, we conclude that $\psi \models \varphi$.

To implement this intuition we first need to duplicate our background knowledge ψ and create its copy ψ' by naming all the atomic propositions A using A'. For each atomic proposition Q that might participate in a query, we then add two more propositions: "$QUERY_Q$" and "$UNKNOWN_Q$". $QUERY_Q$ is used to initiate a query about Q: it will be externally clamped by the user, when he or she inquires about Q. $UNKNOWN_A$ represents the answer of the system. It will be set to TRUE if we can conclude neither that ψ entails φ nor that ψ entails φ.

Our inference engine can be therefore described (using the high-level language of penalty logic) by:

ψ	searches for a model that satisfies also Q
$\cup\psi'$	searches for a model that satisfies also $\neg Q$
$\cup\{< \epsilon, (QUERY_Q \rightarrow Q) >\}$	bias ψ to search for a model that satisfies Q
$\cup\{< \epsilon, (QUERY_Q \rightarrow (\neg Q')) >\}$	bias ψ' to search for a model that satisfies $(\neg Q')$
$\cup\{< \epsilon, (Q \wedge \neg Q') \rightarrow UNKNOWN_Q >\}$	if two satisfying models exist that do not agree on Q, we conclude "UNKNOWN"
$\cup\{< \epsilon, (Q \leftrightarrow Q') \rightarrow (\neg UNKNOWN_Q) >\}$	if despite the bias we are unable to find two such satisfying models we conclude "$\neg UNKNOWN_Q$"

Using the algorithm of Theorem 5, we generate the corresponding network.

The network tries to find models that satisfy also the bias rules. If it succeeds, we conclude "UNKNOWN", otherwise we conclude that all the satisfying models agree on the same truth value for the query. The "UNKNOWN" proposition is then set to "false" and the answer whether $\psi \models \varphi$ or whether $\psi \models \neg\varphi$ can be found in the proposition Q. If Q is "true" then the answer is $\psi \models \varphi$ since Q holds in all satisfying models. Similarly, if Q is false, we conclude that $\psi \models \varphi$

The network converges to the correct answer if it manages to find a global minimum. An annealing schedule like in [Hinton, Sejnowski 86] may be used for such search. A slow enough annealing is certain to find a global minimum and therefore the correct answer, but it might take exponential time. Since the problem is NP-hard, we will probably not find an algorithm that will give us always the correct answer in polynomial time. Traditionally in AI, knowledge representation systems traded the expressiveness of the language they use with the time complexity they allow [Levesque 84]. The accuracy of the answer is usually not sacrificed. In our system we trade the time with the accuracy of the answer. We are given

limited time and we stop the search when this limit is reached. The annealing schedule can be planned to fit the time limitation and an answer is given at the end of the process. Although the answer may be incorrect, the system is able to improve its guess as more time is given.

10. Related work

Derthick [Derthick 88] observed that weighted logical constraints (which he called "certainties") can be used for non-monotonic connectionist reasoning. We follow his direction and there are many similarities and differences that will be discussed in the longer version of this paper. There are however, two basic differences: 1) Derthick's "Mundane" reasoning is based on finding a single most likely model that satisfies a WFF; his system is never skeptical; 2) Our system can be implemented using standard low-order units, and we can use models like Hopfield nets or Boltzman machines that were relatively well studied (e.g., learning algorithms exist).

[Shastri 85] is a connectionist non-monotonic system for inheritance nets that uses evidential reasoning based on maximum likelihood. Our approach is different; we use low-level units and we are not restricted to inheritance networks. Shastri's system is guaranteed to always give the correct answer, whereas we trade the correctness with the time.

Our world rank functions (like $Vrank_{\psi}$ or $Erank_{E}$) have a lot in common with ranked models semantics [Shoham 88], [Geffner 89], [Lehmann 89]. Lehmann's results about the relationship between rational consequence relations and ranked models can be applied to our paradigm; yielding a rather strong conclusion: for every conditional knowledge base we can build a ranked model (for the rational closure of the knowledge base) and implement it as an *Erank* using a symmetric neural net. Also, any symmetric neural net is implementing some ranked model and therefore induces a rational consequence relation.

11. Conclusions

We have shown an equivalence between the search problem of satisfiability and the problem of minimizing connectionist energy functions. Any satisfiable WFF can be described by an n-order energy function with no hidden variables, or by a quadratic function with $O(length(WFF))$ hidden variables. Using the algorithms described we can efficiently determine the topology and the weights of a connectionist network that represents and approximates a given satisfiability problem.

Several equivalent high-level languages can be used to describe symmetric neural networks: 1) quadratic energy functions [Hopfield 82]; 2) high-order energy functions with no hidden units [Pinkas 90a]; 3) propositional logic, and finally 4) penalty logic. All these languages are expressive enough to describe any symmetric network and every sentence of such languages is translatable into a network.

We have developed a calculus based on beliefs augmented by penalties that fits very naturally in the symmetric models' paradigm. This calculus can be used as a platform for defeasible reasoning and inconsistency handling. Some non-monotonic systems can be mapped to this paradigm and therefore suggest settings of the penalties. When the right penalties are given (for example using algorithms like in [Brewka 89] that are based on specificity), our networks features a non-monotonic behavior that (usually) matches our intuition. Penalties do not necessarily have to come from a syntactic analysis of a symbolic language; since those networks can learn, they are capable of adjusting their *Erank* functions and develop their own intuition and knowledge.

We sketched a connectionist inference engine for penalty calculus. When a query is clamped, the global minima of such network correspond exactly to the correct answer. Using massively parallel hardware convergence should be very fast, although the worse case for *correct* answer is still exponential. The mechanism however trades the correctness of the answer with the time given to solve the problem.

References

[Brewka 89] G. Brewka, Preferred sub-theories: An extended logical framework for default reasoning., *IJCAI* 1989, pp. 1043-1048.

[Derthick 88] M. Derthick, "Mundane reasoning by parallel constraint satisfaction", PhD Thesis, TR. CMU-CS-88-182, Carnegie Mellon 1988.

[Geffner 89] H. Geffner, "Defeasible reasoning: causal and conditional theories", PhD Thesis, TR UCLA, 1989.

[Hinton, Sejnowski 86] G.E Hinton and T.J. Sejnowski "Learning and Re-learning in Boltzman Machines" in J. L. McClelland, D. E. Rumelhart *"Parallel Distributed Processing:Explorations in the Microstructure of Cognition"* Vol I pp. 282 - 317 MIT Press 1986

[Hinton89] G.E Hinton "Learning in Mean Field Theory" *Neural Computation* vol 1-1, 1989

[Hölldobler 90] S. Hölldobler, "CHCL, a connectionist inference system for horn logic based on connection method and using limited resources", *International Computer Science Institute* TR-90-042, 1990.

[Hopfield 82] J.J. Hopfield "Neural networks and physical system with emergent collective computational abilities," *Proceedings of the National Academy of Sciences* USA, 1982,79,2554-2558.

[Hopfield 84] J.J. Hopfield "Neurons with graded response have collective computational properties like those of two-state neurons". *Proceedings of the National Academy of Sciences* USA, 1984,Vol 81, pp. 3088-3092.

[Hopfield,Tank85] J.J. Hopfield, D.W. Tank "Neural Computation of Decisions in Optimization Problems" *Biological Cybernetics*, Vol 52, pp. 144-152.

[Lehmann 89] D. Lehmann, "What does a conditional knowledge base entail?", KR-89, *Proc. of the international conf. on knowledge representation*, 1989.

[Levesque 84] H.J Levesque, "A fundamental tradeoff in knowledge representation and reasoning." *Proc. CSCSI-84*, London, Ontario, pp 141-152, 1984.

[Pinkas 90a] G. Pinkas, "The Equivalence of Connectionist Energy Minimization and Propositional Calculus Satisfiability" *Technical Report: WUCS-90-03, Department of Computer Science, washington University* 1990.

[Pinkas 90b] G. Pinkas, "Energy minimization and the satisfiability of propositional calculus", In Touretzky,D.S., Elman, J.L, Sejnowski, T.J., and Hinton, G.E. (eds), *Proceedings of the 1990 Connectionist Models Summer School.* San Mateo, CA: Morgan Kaufmann.

[Pinkas 91] G. Pinkas, " Symmetric Neural Nets and Propositional Logic Satisfiability", To appear in *Neural Computation* Vol 3-2.

[Sejnowski 86] T.J. Sejnowski, "Higher-order Boltzman machines", Neural Networks for Computing", *Proc. of the American Institute of Physics", 151*, Snowbird (Utah) pp 3984, 1986.

[Shastri 85] L. Shastri, "Evidential reasoning in semantic networks:A formal theory and its parallel implementation", *PhD thesis, TR 166, University of Rochester*, Sept. 1985.

[Shastri,Ajjanagadde 90] L. Shastri, V. Ajjanagadde, "From simple associations to systematic reasoning: a connectionist representation of rules, variables and dynamic bindings" TR. MS-CIS-90-05 *University of Pennsylvania, Philadelphia*, 1990.

[Shoham 88] Y. Shoham, "Reasoning about change" *The MIT press*, Cambridge, Massachusetts, London, England 1988.

Reasoning About Disjunctive Constraints

Can A. Baykan and **Mark S. Fox**
Center for Integrated Manufacturing Decision Systems
The Robotics Institute
Carnegie Mellon University
Pittsburgh, PA 15213

31 January 1991

ABSTRACT

We identify a class of constraint satisfaction problems which we call *disjunctive CSPs*. In many domains, such as spatial and temporal reasoning, there exists disjunctive constraints among the variables. A *disjunctive constraint* is a disjunctive combination of *atomic* constraints. Disjunctive constraints pose a problem, because the standard arc consistency algorithms can not deal with disjuncts. When the constraints are in the form of compatibility constraints, it is possible to transform a disjunctive constraint into one without disjuncts. This can not be done if constraints are in any other form. We define disjunctive constraints and disjunctive CSPs, give a path consistency algorithm, and discuss how heuristic measures called *textures* are used to increase efficiency.

1. Introduction

We identify a class of constraint satisfaction problems (CSP) which we call *disjunctive CSPs*. In many domains, such as spatial and temporal reasoning, there exists disjunctive constraints among the variables.

- In spatial layout, topological relations can be expressed as disjuncts of algebraic constraints on lines of the objects [Pfeffercorn 71, Eastman 73, Flemming 78, Baykan & Fox 87].
- In job shop scheduling, multiple orderings of the operations in the process plan, alternative operations, and alternative resources for performing an operation lead to disjuncts [Fox83, Fox90].
- In circuit analysis, the behavior of non-linear devices, such as transistors, can be represented by disjuncts of linear constraints [Stallman & Sussman 77].
- In design, goals can be expressed as disjuncts of lower level constraints [Finger, et al. 91].

Disjunctive constraints pose a problem, because the standard arc consistency algorithms can not deal with disjunctive constraints except when it is possible to combine all disjuncts to form an equivalent conjunctive constraint [Mackworth & Freuder 85]. We define disjunctive constraint satisfaction problems, give a path consistency algorithm, and discuss how they can be solved efficiently by using measures of the topology of the constraint graph called *textures* to order the disjunctive constraints.

2. Definitions

A CSP is made up of variables, values and constraints. The goal is to find all assignments of the values to the variables, such that all constraints are simultaneously satisfied.

There is a set of variables $V = \{v_1, v_2, ..., v_n\}$, each with an associated domain of values. Domains can be *discrete*: $v_i = (1, 2, 3)$ or *continuous*: $v_j = [-200, 350]$; the domain of continuous variable v_j may be a closed interval, defined by a minimum and a maximum value.

A constraint is a relation between some subset of the variables which identifies compatible values of the variables. Constraints can be defined by explicitly listing the compatible values, termed *compatibility* constraints; $c_{ij} = \{(1, 1) (1, 2) (1, 3) (2, 2) (2, 3) (3, 3)\}$ or by equations or inequalities, termed *algebraic constraints*: c_{ij}: $v_i \leq v_j$, or c_{kmn}: $v_k + v_m = v_n$. Compatibility constraints can be defined on variables with finite and discrete domains. Algebraic constraints can be defined on numerical variables with discrete, continuous or interval domains. We will call these compatibility

and algebraic constraints *atomic* constraints, to distinguish them from *disjunctive* constraints defined below.[1]

A *disjunctive constraint* is a disjunctive combination of one or more atomic constraints, such as: $C^{1,2,3}_{ijmnrs}$: $(c^1_{ij}\ c^2_{mn}\ c^3_{rs})$ or C^4_{ij}: (c^4_{ij}). The disjunctive constraint $C^{1,2,3}_{ijmnrs}$ is the combination of the atomic constraints: c^1_{ij}, c^2_{mn} and c^3_{rs}; and is satisfied when any one of its three atomic constraints are satisfied. According to this definition, an atomic constraint is a special type of disjunctive constraint. For example, the disjunctive constraint C^4_{ij} is composed of only c^4_{ij}. Disjunctive constraints may involve arbitrary combinations of disjuncts and conjuncts of atomic constraints, but the canonical form is defined to be *disjunctive normal*, i.e., $C^{1,2,3,4,5}_{ijmnp}$: $((\wedge\ c^1_i\ c^2_{ij})\ (\wedge\ c^3_n\ c^4_{mn})\ c^5_p)$. The top level elements in the domain of a disjunctive constraint in its canonical, disjunctive normal form are called its *disjuncts*. The *disjuncts* of $C^{1,2,3,4,5}_{ijmnp}$ are the three top level elements: $(\wedge\ c^1_i\ c^2_{ij})$, $(\wedge\ c^3_n\ c^4_{mn})$ and c^5_p.

A CSP which contains disjunctive constraints is called a *disjunctive CSP*. Figure 2-1 shows a disjunctive CSP made of discrete variables and compatibility constraints.

3. Where/How Do Disjunctive Constraints Arise?

The questions this section attempts to answer are: Where do disjunctive constraints originate, and how is search controlled to explore this space of alternatives? We will look at two approaches: reductionist approaches as used in constraint satisfaction, and constructive approaches as used in planning.

In the classical CSP formulation, there are no disjunctive constraints [Mackworth 77]. Later extensions to interval constraints required the introduction of disjunctions. Allen's formulation reasons about time intervals and temporal constraints by maintaining a disjunctive set of relations between two time intervals. Propagation eliminates relations from this set as new nodes are added and some relations are removed. The propagation algorithm results in 3-consistency—but not path consistency. Propagation is driven by a transitivity table which lists compatible relations for paths of length 2.

[1] An atomic constraint can be given in arbitrary form (equations, inequalities, tables, logical expressions, or procedures) as long as it can be implemented as a subroutine allowing the checking of whether a set of values satisfies the constraint. The method defined in this paper applies to atomic constraints given in any form.

$v_1 = (1, 2, 3)$

$v_2 = (1, 2, 3)$

$v_3 = (1, 2, 3)$

$c_{12}^1 = \{(1, 2) (1, 3) (2, 3)\}$

$c_{23}^2 = \{(1, 2) (1, 3) (2, 3)\}$

$c_{13}^3 = \{(2, 1) (3, 1) (3, 2)\}$

$c_1^4 = \{1\}$

$c_2^5 = \{1\}$

$C_{123}^{1,2,3} : ((\wedge\ c_{12}^1\ c_{23}^2)\ (\wedge\ c_{12}^1\ c_{13}^3)\ (\wedge\ c_{23}^2\ c_{13}^3))$

$C_{12}^{4,5} : (c_1^4\ c_2^5)$

Assign values to v_1, v_2, v_3 subject to $C_{123}^{1,2,3}$, $C_{12}^{4,5}$

Figure 2-1: A discrete disjunctive CSP

In spatial reasoning, temporal relations are replaced by topological relations, and instead of a single dimension of time, there are two or possibly three dimensions. The objects are defined by intervals in every dimension, instead of being single intervals. Restricting the set of objects to rectangles simplifies this. The relations we want to express, such as adjacency or non-overlap, lead to disjunctions when formulated in terms of algebraic relations between the endpoints of the intervals.

In both temporal and spatial domains, generating a consistent labeling requires that a subset of the disjuncts be selected in order to perform arc-consistency and assign a label.

In the constructive approach, the process of constructing a search path implicitly selects a subset of constraints. Consider the problem space model of problem solving. It assumes that the space of alternatives is too large to be searched. Therefore knowledge in the form of evaluation functions, operator preconditions, etc. is used to limit the paths explored. One of the earlier explicit uses of constraints in heuristic search can be found in the MOLGEN experiment planning system [Stefik 81]. The space of alternative experiment plans is explored hierarchically by first generating an

abstract skeletal plan, then by refining it. During the refinement process, for each plan, constraints are "gathered" and values that satisfy the constraints are identified. Therefore, each plan "selects" a subset of constraints that are relevant to the plan. The ISIS system extended the use of constraints in planning by formalizing the communication between planning levels and the knowledge used to guide search as constraints [Fox 87]. In both cases, there exists a set of constraints, but the subset chosen to satisfy is the byproduct of the plan construction process.

A constructive problem solver is equivalent to the disjunctive CSP formulation described above. In the disjunctive CSP all constraints are given explicitly at the outset, whereas in the constructive approach they are implicit in other decisions.

4. Need for a New Method

The problem is to assign values to all the variables such that all disjunctive constraints are satisfied. A disjunctive constraint is satisfied when one of its disjuncts is satisfied.

A disjunctive constraint composed of compatibility constraints can be transformed into an atomic constraint as follows:

1. Take the union of all the variables of the atomic constraints in the disjunctive constraint. The equivalent atomic constraint will span this set of variables.
2. For each atomic constraint involved, extend each compatibility tuple to include dashes for the new variables.
3. The new compatibility constraint is the union of the compatibility tuples introduced in step two.

For example, let $C^{1,2}_{123}$: $(c^1_{12}\ c^2_{13})$, where c^1_{12} and c^2_{13} are as defined in figure 2-1.

1. The new atomic constraint, $c_{123'} \equiv C^{1,2}_{123}$ is going to cover v_1, v_2 and v_3.
2. Extend the compatibility tuples to cover v_1, v_2 and v_3:
   $c^1_{12} \equiv c_{123''} = \{(1, 2, -)\ (1, 3, -)\ (2, 3, -)\}$
   $c^2_{13} \equiv c_{123'''} = \{(2, -, 1)\ (3, -, 1)\ (3, -, 2)\}$
3. $c_{123'} = (c_{123''} \cup c_{123'''}) = \{(1, 2, -)\ (1, 3, -)\ (2, 3, -)\ (2, -, 1)\ (3, -, 1)\ (3, -, 2)\}$

Thus, it is possible replace a disjunctive constraint with an equivalent atomic constraint, when the disjunctive constraint is composed of compatibility constraints. The disjuncts can be removed from the problem given in figure 2-1 using the above method, resulting in $C^{1,2,3}_{123} \equiv c_{123}$: $\{(1, 2, 3)(2, 3, 1)(3, 1, 2)\}$ and $C^{4,5}_{12} \equiv c_{12'}$:$\{(1, -)(-, 1)$. Then, we can solve it as a regular CSP.

When the atomic constraints are algebraic, we can not remove disjunctions by simplification: i.e., $c_{12}^1 = v_1 \leq v_2$, $c_{34}^2 = v_3 \leq v_4$, and $C_{1234}^{1,2}$: $(c_{12}^1\ c_{34}^2) = (v_1 \leq v_2 \vee v_3 \leq v_4)$. When the domains of variables are discrete, we can transform algebraic constraints into equivalent compatibility constraints. Thus, we can first change continuous variables to discrete, then replace algebraic constraints with compatibility constraints, and remove disjuncts. The drawbacks to this are:

- Sizes of domains can become enormous, depending on the granularity. In a discrete CSP, the complexity of arc-consistency is $O(ea^2)$ [Mohr & Henderson 86], where a is the number of values in the domain of a variable and e is the number of constraints. In an interval CSP, the complexity of arc-consistency is $O(nE)$ where n is the number of variables, and E is the sum of the lengths of all constraints, where the length of a constraint is defined as the number of variables it connects [Davis 87]. It is better to use interval CSPs rather than discrete ones from a complexity viewpoint.
- A coarse grain discretization reduces domain size a, reducing complexity, but it may also eliminate many and in some cases all valid solutions. In scheduling, minutes, hours, days, or weeks can be taken as the smallest grain of time. Using larger units reduces complexity but it may eliminate all solutions.
- For interval variables, node and and arc-consistency is equal to path consistency when there are no loops in the constraint graph [Davis 87]. The complexity of path-consistency is cheaper for continuous CSPs compared to discrete CSPs. The complexity of path consistency for discrete CSPs is $O(n^3a^3)$ where n is the number of variables, and a is the domain size.

When using continuous variables and algebraic constraints, the problem becomes identifying path-consistent combinations of disjuncts that satisfy all disjunctive constraints. Every such combination defines an equivalence class of solutions. Is solving such CSPs problematic? Yes, because preprocessing does not help. Arc-consistency may not significantly reduce the domains of variables when there are disjunctive constraints. Consider arc-consistency given the continuous disjunctive CSP in figure 4-1. The disjunct c_{12}^1 reduces the domain of v_2 = *[50, 100]*, leaves v_1 unchanged, and

---

$v_1 = [50, 100]$

$v_2 = [1, 100]$

$v_3 = [1, 50]$

$c_{12}^1 = v_1 \leq v_2$

$c_{23}^2 = v_2 \leq v_3$

$C_{123}^{1,2} : (c_{12}\ c_{23})$

Figure 4-1: A continuous disjunctive CSP

does not affect v_3 because it does not span that variable. The disjunct c_{23}^2 reduces the domain of

v_2 = *[1, 50]*, leaves v_3 unchanged, and does not span v_3. Every value of v_2 has supporting values in either v_1 or v_3. Arc-consistency with respect to $C_{123}^{1,2}$ can not remove any values from the domains of v_1, v_2 and v_3.

How do we find a feasible solution? We can replace a disjunctive constraint by one of its disjuncts, and use the atomic constraints to revise the domains of variables. This must be carried out for all disjunctive constraints. Each combination of disjuncts is another CSP.

Assume that the there are n disjunctive constraints, each containing b disjuncts. The number of possible combinations containing all disjunctive constraints, i.e., candidate solutions, is b^n. Partial solutions can be composed of any combination of the *1...n* disjunctive constraints. The number of partial solutions containing m disjuncts is $(n!/(n-m)!m!)b^m$. The number of all combinations, containing any number of disjunctive constraints from 1 to n is $\sum_{m=1}^{n}(n!/(n-m)!m!)b^m$. When propagating at partial solutions, adding a new disjunct to an alternative changes not only the variables covered by the disjunct but may change all the variables connected together by the set of disjuncts due to constraint propagation.

- A problem solver which enumerates candidate solutions, and evaluates only complete solutions will evaluate b^n alternatives.
- An ATMS searches through the space of partial solutions in breadth first fashion, looking at all combinations of *1*, then all combinations of *2, 3, ..., n* disjunctive constraints [de Kleer 86]. It identifies minimal sets of inconsistent combinations, and avoids repeating these in larger sets.
- Backtracking tries one ordering of the disjunctive constraints in each path. This is possible because the order in which disjunctive constraints are considered does not change the set of solutions, although it affects search efficiency. A dynamic ordering scheme may try a different order in different branches of the search tree.
- Backtracking can be combined with an ATMS. Such a system can either evaluate only complete solutions as in the assumption based DDB, and in each candidate solution consider every possible combination of *1, ..., n* disjuncts [de Kleer & Williams 86]; or evaluate all partial solutions along the search path as in EL [Stallman & Sussman 77]. In both cases, the ATMS keeps track of every propagation step to identify the minimal set of disjuncts causing an inconsistency, and avoids repeating the same propagation steps in different branches of the search tree by indexing them.

What are the trade offs between the four approaches given above, when we want all solutions to a disjunctive CSP? Both backtracking and ATMS approaches are better than complete enumeration, because they use a failing partial solution to eliminate all complete solutions derived from it. The ATMS does extra work by searching a much larger space than backtracking, and searching parts of it which are not reached by backtracking [de Kleer & Williams 86]. Backtracking does extra work

due to thrashing, where the same inconsistency is detected many times in different search paths [Mackworth 77]. Ordering disjunctive constraints may reduce thrashing, but is not guaranteed to eliminate it. Combining ATMS with backtracking eliminates thrashing and repeating the same inferences in different search paths. Its efficiency depends upon the efficiency of the backtracking algorithm which controls it.

The backtracking algorithm given below can be combined with an ATMS, which changes it from sequential to dependency-directed backtracking.

5. Solution Method

The method we propose uses CSP formulation recursively, and combines search and arc-consistency. There exists interesting heuristics and machinery for selecting variables and values, which we can use to solve regular CSPs. The method we propose is only reasonable if there exists good heuristics for ordering disjunctive constraints. In the rest of this section, we describe this perspective and the heuristics which guide search.

In order to solve a disjunctive CSP, each disjunctive constraint is treated as a variable. Their domains consist of the disjuncts, which are atomic constraints in disjunctive normal form, as defined in section 2. This can be thought of as constructing the dual of the disjunctive CSP that is not disjunctive. The steps for solving the disjunctive CSP are as follows:

1. Select a search state to expand. If there are no active states, stop.
2. Select a disjunctive constraint. If there are no active disjunctive constraints in state, the state is a solution, go to step 1.
3. For every disjunct in the domain of the selected disjunctive constraint, create a new search state. In each new state do:
4. Achieve node and arc consistency with respect to the disjunct(s) by incremental constraint satisfaction.
5. If the domain of a variable becomes empty, eliminate state. The disjunct(s) added last is inconsistent with the previous ones.
6. Check the active disjuncts in the domains of future disjunctive constraints, and remove those that are inconsistent with the previous ones, or those that are violated due to the reduced domains of the variables.
7. If all disjuncts are removed from the domain of a future disjunctive constraint, the constraint is violated. Eliminate the state, or use violated constraint to rate state.
8. If some future disjunctive constraints have only one active disjunct left in their domains, select them and go to step 4.

9. Go to step 1.

In the disjunctive CSP algorithm given above, selecting a disjunctive constraint (step 2) corresponds to variable selection in solving CSPs by backtrack, and the disjuncts in its domain correspond to the values of the variable. When we want all solutions, all disjuncts must be tried, thus disjunct selection is not an issue. The loop consisting of steps 4—8 corresponds to forward checking in a regular CSP. Step 4 of the algorithm uses the *procedure Waltz*, modified to handle the addition of the disjuncts selected in each state [Davis 87;p.286-287]. Step 5 eliminates the current state from further consideration, if the domain of a variable becomes empty during propagation. In disjunctive CSP with variables that have interval domains and no loops in the constraint graph, step 5 does not occur because inconsistent disjuncts are removed in step 6 of the previous state. Step 6 checks the active disjuncts in the domains of future disjunctive constraints to find out whether any of them are satisfied or violated as a result of reducing the domains of variables in step 4. Step 6 can be thought of as an extension of propagation: selecting a disjunct changes the constraint graph in that state, and reduces the domains of variables, this in turn may cause future disjuncts to be satisfied or violated. Since one of the disjuncts of every disjunctive constraint must be satisfied, step 7 eliminates a state where all disjuncts in a disjunctive constraint are violated. Step 8 satisfies any disjunct that is the only active one remaining in the domain of a future disjunctive constraint. Steps 7 and 8 are applied recursively in every state to reduce search. This algorithm combines search and arc-consistency. Search tries combinations of disjuncts. Arc-consistency eliminates values from domains of variables and disjuncts from the domains of disjunctive constraints. Each solution state is a combination of disjuncts; it defines an equivalence class of solutions. Individual solutions can be found by selecting a unique value from the domain of every interval variable.

Figure 5-1 shows the steps of the disjunctive CSP algorithm, when it is used to solve the problem given in figure 2-1. The numbers on the left of each line show the steps of the algorithm, and the rest of the line shows the operations carried out in that step. The problem is solved in two states.

Search efficiency can be measured in terms of:

- total time,
- total number of propagation operations (in step 4 of the algorithm) and the total number of tests (in step 6 of the algorithm),
- number of search states expanded.

1. ***state*** ← *initial-state*
2. ***constraint*** ← $C_{12}^{4,5} : (c_1^4\ c_2^5)$

3. ***disjunct*** ← c_1^4, *create state1*
4. $v_1 = (1)$
6. $C_{123}^{1,2,3} : ((\wedge\ c_{12}^1\ c_{23}^2))$
8. $(\wedge\ c_{12}^1\ c_{23}^2)$
4. $v_2 = (2), v_3 = (3)$

3. ***disjunct*** ← c_2^5, *create state2*
4. $v_2 = (1)$
6. $C_{123}^{1,2,3} : ((\wedge\ c_{23}^2\ c_{13}^3))$
8. $(\wedge\ c_{23}^2\ c_{13}^3)$
4. $v_1 = (3), v_3 = (2)$

1. ***state*** ← *state2*
2. ***constraint*** ← ∅, *state2* is a solution
1. ***state*** ← *state1*
2. ***constraint*** ← ∅, *state1* is a solution
1. ***state*** ← ∅, STOP

Figure 5-1: Steps of the disjunctive CSP algorithm for solving the disjunctive CSP in figure 2-1.

Search efficiency depends on the choice of a disjunctive constraint in step 1. The order states are expanded does not matter if we are satisficing and looking for all solutions to the problem. Since we can not use disjunctive constraints directly to reduce the domains of variables, we try each of their disjuncts separately. This gives rise to a combinatorial explosion which must be controlled. The operation of the algorithm consists of two phases: divide and simplify.

The leftmost diagram in figure 5-2 shows the problem space labeled P, and the solution space labeled R. For the purposes of the following discussion, assume that the disjuncts in the domain of a

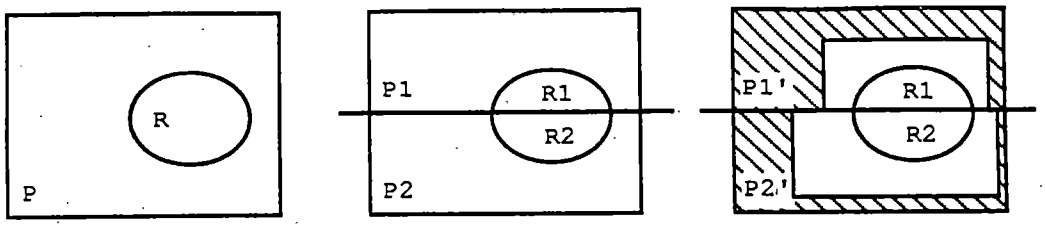


Figure 5-2: Problem space and the divide and simplify operations

disjunctive constraint are mutually exclusive. Selecting a disjunctive constraint D_i in step 2 of the algorithm, and trying each of its disjuncts in a different search state partitions the problem space into n disjoint parts, where n is the number of active disjuncts in the domain of C_i. This is the **divide** step, seen in the middle of figure 5-2. In each state, the disjunct that is selected reduces the problem space by constraint propagation, carried out in steps 3—7 of the algorithm in a least commitment mode. Propagation reduces the domains of variables, and then testing and subsequent steps reduces the domains of disjunctive constraints. This mode of operation is shown at the right in figure 5-2.

Given that the algorithm operates as described above, how can we make the algorithm operate efficiently? We can identify two types of division of the problem space. A *heterogeneous* division

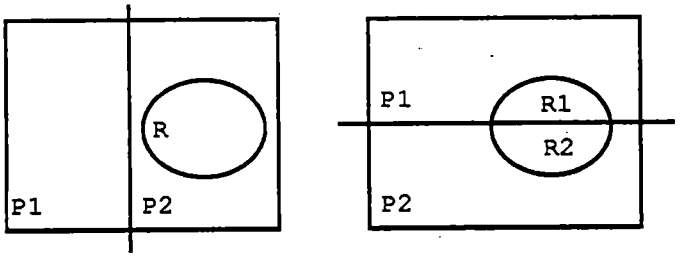


Figure 5-3: Homogeneous and heterogeneous divisions of the problem space

partitions the problem space into portions that all contain some solutions, whereas a *homogeneous* divider partitions the search space into a region which contains all solutions and other regions that contain none. Thrashing behavior occurs while searching in regions that contain no solutions. We can avoid looking at such portions, because the disjuncts defining empty regions may be eliminated as a result of selecting heterogeneous dividers and achieving arc-consistency. Also, disjuncts which eliminate large portions of the problem space are more useful. Disjuncts which extend the constraint graph to span new variables are more useful, as are disjunctive constraints with fewer active disjuncts.

We use *problem textures* [Fox, et al. 89] which are measures of problem topology to help focus search in a way that it is solved more efficiently. Textures are used for selecting a disjunctive constraint in step 1 of the algorithm given above. Textures can be on variables, atomic constraints and disjunctive constraints. The textures we have been using are:

- *T1*: This texture is a measure of disjunctiveness. It's value is the number of active disjuncts remaining in the domain of a disjunctive constraint. It selects a constraint that has fewer disjuncts.
- *T2*: This texture is a measure of the overlap of atomic constraints among disjunctive constraints. An atomic constraint may appear in more than one disjunctive constraint. When that is the case, satisfying it will satisfy multiple disjunctive constraints. T2 picks a disjunctive constraint sharing the largest number of atomic constraints with others.[2]
- *T3*: measures the looseness of disjunctive constraints. It looks at the resulting domain sizes of the variables of an atomic constraint. T3 selects a disjunctive constraint that is composed of atomic constraints that severely limit the domains of their variables.
- *T4*: measures the looseness of variables. It looks at the number of atomic constraints on a variable. It selects a disjunctive constraint that is on variables having a large number of constraints.

There are two problems that must be resolved in using textures to select a disjunctive constraint. The first is that textures on variables and atomic constraints must be combined to result in a value for each texture on a disjunctive constraint. The second is how to use the four textures on each disjunctive constraint to select among different disjunctive constraints.

Lexicographic ordering of the textures is used for selecting a disjunctive constraint. Each texture eliminates some disjunctive constraints. If more than one remains after the application of a texture, the next texture is used. If multiple disjunctive constraints remain after applying all textures, one is selected at random. We have been experimenting with different orderings of the textures.

If we want only the first solution, then ordering the disjuncts also affects efficiency. T2, T3 and T4 can be used to order the disjuncts. This corresponds to value selection heuristics in regular CSPs.

[2]Some atomic constraints may be relaxations of others, or satisfy other atomic constraints due to transitivity. It is computationally more expensive to detect these cases.

6. Experimental Results

We have formulated spatial layout as a disjunctive CSP and disjunctive COP, and solved problems involving the layout of work stations and work centers in a manufacturing facility, rooms in a house, appliances and cabinets in a kitchen, and bin packing problems with rectangular blocks [Baykan & Fox 89, Baykan & Fox 87].

Textures reduce search. Compared to random selection of variables, using combinations of textures reduced search states by 70% in kitchen problems and 84% in blocks problems. Figure 6-1 shows the number of search states required for finding all solutions to five kitchen layout problems, under different combinations of texture measures. The combinations tested are:

- *method 0:* select a constraint at random,
- *method 1:* T4,
- *method 2:* T1,
- *method 3:* T4 and T1,
- *method 4:* T4, T1 and T3.

When a combination of measures is used, they are applied in the order: T4, T1, T3. Each measure eliminates some constraints from consideration. If more than one constraint remains after applying the texture measure(s), specified by the method, a constraint is selected at random. The number of states given for each problem-method combination is the average of three runs. In the second problem, method 4 reduces search by more than 80% compared to method 0, and in the third problem by 35%. These results were reported in [Fox, et al. 89, Baykan & Fox 89]. The order of

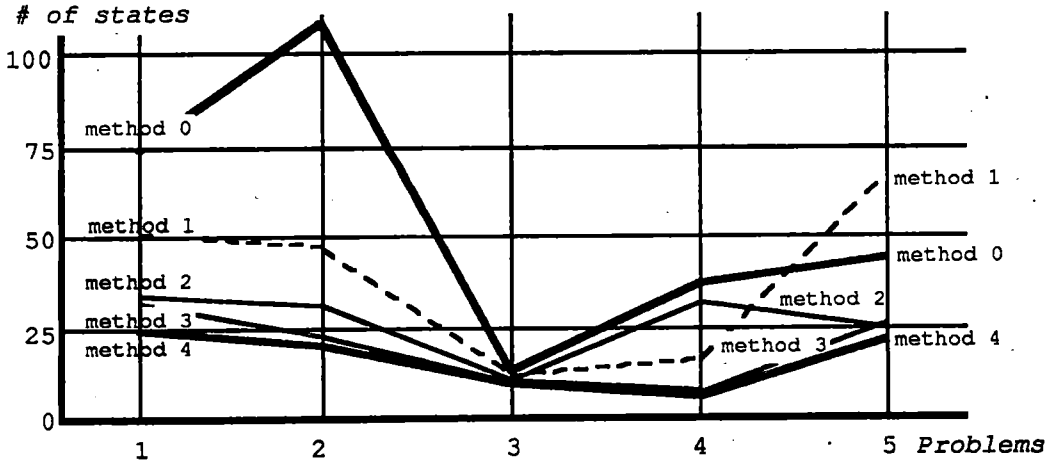


Figure 6-1: Effectiveness of texture measures in reducing search

applying the textures has a significant effect on search efficiency. We are experimenting with new textures, application orders, and combinations of textures.

7. Role of ATMS

We are exploring with Reid Simmons, using the Quantity Lattice program combined with an ATMS [Simmons 86], how an ATMS would solve this problem; and hope to have experimental data by the time of the workshop.

8. Conclusion

We identify a class of constraint satisfaction problems called disjunctive CSPs, and give an algorithm which is especially useful when the problem has continuous variables and algebraic constraints. This is a backtracking algorithm which uses two special heuristics: it satisfies a disjunctive constraint that has a single alternative left, and it eliminates a state when a disjunctive constraint has no possibility of being satisfied. It relies on ordering the disjunctive constraints for efficient backtracking. We define the characteristics leading to an efficient order, and give a set of textures, which are simple heuristic measures for dynamically ordering the constraints. This algorithm can also be combined with an ATMS to do dependency-directed backtracking. The performance of textures have been reported elsewhere. We are collecting more data about the performance of textures both with the sequential backtracking algorithm given, and with a dependency-directed backtracking algorithm that results when an ATMS is combined with the backtracking algorithm given.

References

[Baykan & Fox 87]
Baykan C.A. and Fox M.S.
An Investigation of Opportunistic Constraint Satisfaction in Space Planning.
In *Proceedings of IJCAI-10*, pages 1035-1038. IJCAI, August, 1987.

[Baykan & Fox 89]
Baykan C. and Fox M.S.
Constraint Satisfaction Techniques for Spatial Planning.
In *Preliminary Proceedings of the Third Eurographics Workshop on Intelligent CAD Systems*, pages 211-227. CWI, Amsterdam, 1989.

[Davis 87] Davis E.
Constraint propagation with interval labels.
AI 32(3):281-331, July, 1987.

[de Kleer 86] de Kleer, J.
An Assumption-based TMS.
AI 28:127-162, 1986.

[de Kleer & Williams 86]
De Kleer, J. and Williams, B.
Back to backtracking: Controlling the ATMS.
In *Proceedings of AAAI-86*, pages 910-917. Morgan Kaufmann Publishers Inc., 1986.

[Eastman 73] Eastman C.M.
Automated space planning.
AI 4:41-64, 1973.

[Finger, et al. 91] Finger, S., Fox, M.S., Prinz, F.B., Rinderle, J.R.
Concurrent Design.
Applied Artificial Intelligence (Special Issue on AI in Manufacturing), 1991.

[Flemming 78] Flemming U.
Wall representations of rectangular dissections and their use in automated space allocation.
Environment and Planning B 5:215-232, 1978.

[Fox 87] Fox M.S.
Constraint-Directed Search: A Case Study of Job-Shop Scheduling.
Morgan Kaufmann Publishers, Inc., 1987.

[Fox, et al. 89] Fox M.S., Sadeh N., and Baykan C.
Constrained heuristic search.
In *Proceedings of IJCAI-11*, pages 309-315. IJCAI, 1989.

[Mackworth 77] Mackwoth A.K.
Consistency in networks of relations.
AI 8:99-118, 1977.

[Mackworth & Freuder 85]
Mackworth A.K., and Freuder E.C.
The complexity of some polynomial network consistency algorithms for constraint satisfaction problems.
AI 25:65-74, 1985.

[Mohr & Henderson 86]
Mohr R. and Henderson T.C.
Arc and path consistency revisited*.
AI 28:225-233, 1986.

[Pfeffercorn 71] Pfeffercorn C.
Computer Design of Equipment Layouts Using the Design Problem Solver.
PhD thesis, Carnegie-Mellon University, May, 1971.

[Simmons 86] Simmons, R.
Commonsense Arithmetic Reasoning.
In *Proceedings of AAAI-86*, pages 118-124. Morgan Kaufmann Publishers Inc., 1986.

[Stallman & Sussman 77]
Stallman M.R. and Sussman G.J.
Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis.
AI 9(2):135-196, October, 1977.

[Stefik 81] Stefik M.J.
Planning with constraints (MOLGEN: Part 1).
AI 16(2):111-140, May, 1981.

USING NETWORK CONSISTENCY TO RESOLVE NOUN PHRASE REFERENCE

Nicholas J. Haddock
Hewlett-Packard Laboratories
Filton Road, Stoke Gifford,
Bristol BS12 6QZ, U.K.
Email:njh@hpl.hp.co.uk

Abstract

This paper investigates a problem of natural language processing from the perspective of AI work on constraint satisfaction. We argue that for a class of referring expression the task of noun phrase reference evaluation corresponds to an "easy" constraint satisfaction problem, and can thus be performed by low-power network consistency techniques. To illustrate, we introduce a linguistic fragment which provably generates tree-like constraint problems. This enables us to infer that strong arc consistency is sufficient to resolve this class of expression. The paper concludes by pointing to semantic phenomena in English which determine constraint problems with a more complex connective structure.

1 Introduction

This paper addresses a problem in computational linguistics from the perspective of AI work on constraint satisfaction. We are interested in the process of evaluating singular noun phrases which refer to known entities in a context. Suppose that during a discourse, a speaker instructs a hearer to

(1) Get me *the green apple on the table*

In order to fulfill this instruction, the hearer must evaluate, or "resolve", the reference of the noun phrase *the green apple on the table* to some specific entity known to him or her. The present paper reviews how the task of resolving reference may be cast as a constraint satisfaction problem, and goes on to explore the use of network consistency techniques for processing such reference-oriented constraint problems. These techniques are generally less powerful than search procedures like backtracking, but have a number of desirable properties, including efficiency. The main purpose of the paper is to show that an extremely limited set of network consistency operations is sufficient to resolve the reference of an illustrative, generatively defined class of definite noun phrase.

2 Reference as a Constraint Satisfaction Problem

Mackworth (1977a) defines a *constraint satisfaction problem* (CSP) as a set of *variables*, each of which must be instantiated in a particular *domain* of values, and a set of *constraints* which the values of the variables must simultaneously satisfy. A CSP can be schematised as the formula in (2)

(2) $(\exists x_1)(\exists x_2)\ldots(\exists x_n)(x_1 \in D_1)(x_2 \in D_2)\ldots(x_n \in D_n)P(x_1, x_2, \ldots, x_n)$

in which each variable x_i is associated with a domain D_i, and where $P(x_1, x_2, \ldots, x_n)$ abbreviates a conjunction of constraints on subsets of the variables. A *solution* to a CSP is an

assignment of values $< a_1, a_2, \ldots, a_n > \in D_1 \times D_2 \times \ldots D_n$ to the variables $< x_1, x_2, \ldots, x_n >$ which simultaneously satisfies all the constraints.

In order to characterise the problem of contextual reference, we must first elaborate on the notion of context. We regard a *context* as consisting of those entities and relations which have been made salient in the hearer's mind, through either linguistic or non-linguistic means. For our purposes it is adequate to represent the hearer's contextual knowledge as a finite set of first-order predications, such as the following:

(3) {man(man1), man(man2), town(town1), town(town2), town(town3), river(river1), river(river2), near(town1,river1), near(town2,river1), visit(man1,town1), visit(man1,town2), visit(man2,town3)}

For example, this indicates that there are two men, two rivers and three towns, that *man*1 visits *town*1, and so on.

Consider the problem of determining the contextual entities involved in the reference of the following noun phrase (assuming that the prepositional phrase *near a river* relates to the town, rather than the man or the visiting event, and ignoring tense):

(4) the man who visited a town near a river

The use of the definite article *the* in (4) is appropriate if (a) there is a man who visits a town near a river, and (b) there is only one such man in the context. The first criterion can be succinctly expressed as a CSP in the style of (2):

(5) $(\exists x_1, x_2, x_3)(x_1 \in D_1)(x_2 \in D_2)(x_3 \in D_3) man(x_1) \wedge visit(x_1, x_2) \wedge town(x_2) \wedge near(x_2, x_3) \wedge river(x_3)$

In words, there exists an x_1, x_2, and x_3, in specified domains, such that x_1 is a man, x_2 is a town and x_3 is a river, and x_1 visits x_2, and x_2 is near x_3. We will assume that all variables in our reference-oriented CSPs start out with the same value domain, namely the set of all entities in the context. So, in the context of (3),

$$D_1 = D_2 = D_3 = \{man1, man2, town1, town2, town3, river1, river2\}.$$

Satisfaction of the formula in (5) will involve assigning x_1, x_2 and x_3 values from this set, and seeing whether the instantiated constraints coincide with the formulae in the context. The second criterion involved in establishing the reference of a definite noun phrase (NP) concerns uniqueness in the context. For (5), this can be seen as a meta-level check that all solutions to the CSP instantiate x_1 to the same entity.

3 Network Consistency

There are a variety of procedures available to solve or partially solve CSPs. Our interest is in the role of a class of network consistency algorithms discussed extensively in the literature (Mackworth, 1987). These algorithms view the CSP as an annotated graph known as a *constraint network*. Given this representation, Montanari (1974), Mackworth (1977a), Freuder (1978), Dechter and Pearl (1988) and others have defined various states of consistency in a constraint network, including *node consistency*, *arc consistency*, and *path consistency*. Here, we will say that a constraint network is *strongly arc consistent* iff it is node and arc consistent. Similarly, a constraint network is *strongly path consistent* iff it is node, arc and path consistent.

Node and arc consistency can be enforced by a progressive operation of *domain refinement*. Assuming the context of (3), the network for (5) could be made could be made strongly arc consistent by an algorithm such as Mackworth's (1977a) AC-3. This would return the following refined domains:

$$D_1 = \{man1\}, D_2 = \{town1, town2\}, D_3 = \{river1\}$$

4 The Adequacy of Network Consistency for Reference Evaluation

Network consistency techniques are often used to reduce the search space posed by a CSP before the invocation of a search procedure like backtracking. In the case of the above example, the strongly arc consistent network allows us to infer that the set of solutions is some subset of the cross-product of the refined domains. But an interesting characteristic of the reference problem is that the felicity of a definite NP can be determined without computing the actual tuples of values which solve its constraint problem. All we require is that the procedure for reference evaluation is capable of associating each variable in the constraint problem with a set of entities, such that every entity in each variable's set participates in some solution to the problem. The referential uniqueness of a variable in a CSP can then be confirmed by a simple, meta-level check that its domain is a singleton set. In view of this requirement, we borrow the following definition from Dechter et al. (1989):

Definition 1 A value a in a domain D_i is *feasible* if there exists a solution to the constraint problem in which the variable x_i is instantiated to a. The set of feasible values of a variable is its *minimal domain.*

Under what circumstances does network consistency guarantee minimal domains? Domain minimality can be shown to follow from Freuder's (1982) condition for backtrack-free search, which relates the degree of network consistency to the connective structure of the constraint graph. Here we enumerate three special cases which follow from Freuder's theorem:

Corollary 1 *The domains of a constraint graph are minimal if:*
(a) the constraint graph has width 0, and it is node consistent;
(b) the constraint graph has width 1, and it is strongly arc consistent;
(c) the constraint graph has width 2, and it is strongly path consistent.

Following Freuder (1982), a constraint graph has width ≤ 1 iff it is a forest. There is a high premium on a CSP having a tree-structured constraint graph, since strong arc consistency can be achieved in $O(nk^2)$, where n is the number of variables, and k is the size of each value domain (Dechter and Pearl, 1988). Moreover, unlike higher degrees of consistency, node and arc consistency can be achieved without changing the width of the constraint graph. Hence, the application of a strong arc consistency algorithm to a tree-structured network will always yield minimal domains.

It is envisaged that much of the semantic structure of English is tree-structured. Simple nouns, and intersective adjectives such as *red*, translate into unary constraints; prepositions correspond to binary constraints, and verbs to n-ary constraints, where $n \geq 1$. Moreover, in the deep structure of the language, these semantic predicates tend to be strung together in a linear, sparse fashion. By way of re-inforcing this point, the following section introduces a linguistic fragment which provably generates only tree-structured semantic forms. The subsequent section discusses semantic phenomena which take us beyond tree-structure.

5 A Tree-Structured Fragment

In common with many natural language systems, our fragment will adopt an essentially *compositional* style of analysis, in which the semantic translation of a linguistic constituent is a simple combination of the semantic translations of its subconstituents (Schubert and Pelletier, 1982). We also assume that this compositional process of semantic translation is driven by the application of syntactic rules to the input string.

All constituents are associated with a four-part representation, comprising (1) a syntactic category, (2) a semantic variable, or tuple of variables, (3) a list of constraints (the *constraint list*), and (4) a list of meta-level conditions on the cardinalities of specified domains (the *cardinality list*). For example, the lexicon contains the following entry for *lake*:

(6) $lake := \mathsf{N}_x : [lake(x)] : []$

This entry defines (":=") *lake* as a noun (N) associated with the semantic variable x, notated as a subscript. The word is associated with the single constraint $lake(x)$, and does not introduce any cardinality conditions. Square brackets are used to indicate lists, as in Prolog, and a list of more than one constraint should be read as a conjunction of constraints. All variables in a constraint list are assumed to be existentially quantified, and, moreover, all variables appearing in the representation of a constituent are *local* to that constituent.

A preposition is defined in a similar manner:

(7) $near := \mathsf{P}_{<x,y>} : [near(x,y)] : []$

Thus, the preposition (P) *near* introduces the binary constraint $near(x,y)$ and, again, no cardinality conditions. However, in contrast to (6), the constituent is relational. We therefore attach the tuple $< x, y >$ to its syntactic category, where x associates with the prepositional subject and y with the prepositional object.

In order to illustrate a syntactic-semantic rule, assume that the rule set has already analysed the definite NP *the lake* as follows:

(8) $the\ lake := \mathsf{NP}_x : [lake(x)] : [|D_x| = 1]$

Hence, *the lake* has been analysed as an NP associated with the variable x, the constraint $lake(x)$, and the cardinality condition that the domain of x should be a singleton, $|D_x| = 1$. The PP *near the lake* can now be analysed by the following augmented phrase-structure rule:

(9) $\mathsf{PP}_x : (C_1 + C_2) : (S_1 + S_2) \longrightarrow \mathsf{P}_{<x,y>} : C_1 : S_1 \quad \mathsf{NP}_z : C_2 : S_2 \quad \text{where } \{y = z\}$

The rule in (9) combines a preposition with an NP on its right to form a PP. The constraint list of the PP constituent is formed by appending the constraint lists of its two subconstituents, C_1 and C_2; we notate this operation as $C_1 + C_2$. The same operation is performed on the subconstituents' cardinality lists, S_1 and S_2.

The rule above can thus be seen as conjoining two separate CSPs to form a larger, composite CSP. By itself this operation will produce no variable-connections between the component CSPs. The responsibility for connective structure depends solely on the manner in which the rule manipulates the semantic variables subscripted on syntactic categories. In (9), the rule *unifies* the variable associated with the prepositional object with the variable associated with the NP. For clarity, we extract all such variable unifications to the right, in the form of a *where*-clause.[1] Hence, the application of (9) to the constituents defined in (7) and (8) yields the following representation for the PP,

(10) $near\ the\ lake := \mathsf{PP}_x : [near(x,y), lake(y)] : [|D_y| = 1]$

in which the variable y from (7) has been unified with the variable x from (8), and is now named y.

A full version of this paper (Haddock, 1991) presents the complete set of augmented phrase-structure rules and lexical entries, NPG1, which treats simple examples of adjectival modification, and complex NPs involving relative clauses and PPs. Two example NP analyses are given below (ignoring tense):[2]

[1] A more common practice is to encode such term unifications directly into the rule (Pereira and Shieber, 1987). However, in the present circumstances this would be somewhat opaque.

[2] Note first that this model gives any uniqueness condition "wide-scope" over the entire CSP of the noun phrase. This is not an essential feature of the system, and other strategies for the discharge of the condition are possible, such as the "narrow-scope" interpretation implemented by Pereira and Pollack (forthcoming). Second, although our model has been devised mainly for the purposes of this paper, Haddock (1988) shows

(11) a the man who bought a book

$\mathsf{NP}_x : [man(x), buy(x,y), book(y)] : [|D_x| = 1]$

b the green cup on the table

$\mathsf{NP}_x : [green(x), cup(x), on(x,y), table(y)] : [|D_x| = 1, |D_y| = 1]$

The fact that our fragment NPG1 generates only CSPs with tree-like constraint graphs hinges on the nature of the unifications performed in the *where*-clauses. First, we observe a fact which can be informally stated as follows:

Lemma 1 *If G' and G'' are trees, then the graph which results from unifying a single vertex $u \in V(G')$ with a single vertex $v \in V(G'')$ is also a tree.*

This follows straightforwardly from the fact that a tree is a connected graph with $n-1$ edges, where n is the number of vertices. We can then state our theorem about NPG1:

Theorem 1 *The constraint lists generated by the linguistic fragment* NPG1 *are tree-like.*

Theorem 1 can be shown to be proven from the following observations: (a) all lexical entries have tree-like constraint lists; (b) semantic variables are local to each lexical entry, and to each rule; and (c) each rule unifies at most one variable from its left-hand daughter with one variable from its right-hand daughter.

Once the NP has been analysed, its constraint list can be transformed into a network and subjected to strong arc consistency. Once the network is consistent, the cardinality conditions are evaluated with respect to the refined domains. If either of these phases should fail, the NP is rejected as infelicitous, an action which can help resolve structural ambiguities encountered by the syntactic parser (Crain and Steedman, 1985). We conclude this section with an obvious corollary of Corollary 1, Theorem 1 and the aforementioned results on the complexity of strong arc consistency. Given that it is easy to establish whether a set is a singleton, the following holds for the kind of reference evaluation investigated in this paper:

Corollary 2 *Once* NPG1 *has derived the semantic translation for an NP in the input string, the reference of that NP can be resolved in $O(nk^2)$ steps.*

6 Beyond Tree Structure

This section briefly illustrates semantic phenomena which introduce cycles in the constraint graph. Cycles may be introduced on at least three distinct levels of semantic interpretation. Here we give one example at each level, although these are not the only instances. However, it is hypothesised that such cyclic expressions are in the minority.

Lexical semantics It is common to decompose lexical predicates into finer-grained predicates which correspond to the underlying knowledge representation. At least in principle, these sub-linguistic constraints may have an arbitrary connective structure. To take a simple example, the lexical constraint $over(x,y)$ may map into $above(x,y) \wedge \neg contact(x,y)$, which is cyclic (albeit simple to eliminate).

Structural semantics Certain syntactic-semantic rules can induce a kind of "double-binding" in the semantic translations. This occurs in the noun phrase in (12a), for example, which contains what Engdahl (1983) calls a "parasitic gap". The CSP for (12a) is shown in (12b).

how a similar scheme of semantics may be incrementally evaluated by network consistency during categorial grammar parsing.

(12) a an apple which a man ate without peeling

b $(\exists x_1, x_2, x_3)(x_1 \in D_1)(x_2 \in D_2)(x_3 \in D_3)\mathit{event}(x_1) \wedge \mathit{man}(x_2) \wedge \mathit{apple}(x_3) \wedge \mathit{eat}(x_1, x_2, x_3) \wedge \mathit{withoutpeeling}(x_1, x_3)$

So there exists an event x_1 in which a man x_2 eats an apple x_3 and, furthermore, the event x_1 is carried out without peeling the apple x_3. It is not difficult to devise contexts for this CSP which would thwart an appropriate strong arc consistency algorithm, such as Mackworth's (1977b) algorithm for n-ary constraints.

Anaphoric semantics Consider the indefinite NP in (13a):

(13) a a man who visits a town near his birth-place

b $(\exists x_1, x_2, x_3)(x_1 \in D_1)(x_2 \in D_2)(x_3 \in D_3)\mathit{man}(x_1) \wedge \mathit{visit}(x_1, x_2) \wedge \mathit{town}(x_2) \wedge \mathit{near}(x_2, x_3) \wedge \mathit{birthplace}(x_3, x_1)$

On the assumption that (13a) refers non-specifically to some man known to the hearer, and depending on the state of the discourse, the possessive pronoun *his* may refer either to some male entity salient in the hearer's discourse model or be bound to the reference of the entire complex noun phrase in which it is embedded. In the latter case it becomes an instance of what Partee (1978) and others have called "bound-variable anaphora". To see why, consider the CSP-style semantic translation for the bound-variable interpretation of (13a), in (13b). Here, a man x_1 visits a town x_2 which is near the birthplace x_3 of the man x_1, whoever he is; the variable representing the reference of the *male person whose birthplace it is* has been bound to the variable representing the reference of *a man who visits ....* The CSP in (13b) corresponds to a complete width-2 constraint graph, and thus path consistency is necessary (and sufficient) to evaluate its reference.

7 Related Work and Conclusion

Network consistency techniques have been applied to a variety of problems in natural language processing, including morphological analysis (Barton, Berwick and Ristad, 1987; Barton, 1986), form-class disambiguation (Duffy, 1986), parsing (Maruyama, 1990), word-sense disambiguation (Winston, 1984), and natural language generation (Dale and Haddock, forthcoming). Mellish (1985), Rich, Wittenburg, Barnett and Wroblewski (1987), and Haddock (1988, 1989) have used network consistency to tackle various aspects of reference evaluation.

The published accounts of these investigations rarely discuss the issue of the sufficiency of network consistency for the task in hand. However, Mellish and Barton do raise the question of adequacy, and both make the empirical observation that their network consistency algorithm seems to be adequate in their problem arena (in both cases, approximate forms of strong arc consistency are used). Barton further hypothesises that natural-language problems may have a special modular, separable nature which makes them amenable to such techniques, but in neither case is the discussion related to a formal notion of adequacy, such as that provided by Freuder (1982).

Against this background, the main developments reported in this paper are (1) the specification of a linguistic fragment which provably generates tree-like CSPs, and (2) the observation that minimal domains are a sufficient interface to the results of reference evaluation for a class of definite NP. Taken together, these enable us to conclude that strong arc consistency is adequate to evaluate this class of referring expression.

The model NPG1, and the assumed evaluation procedure of network consistency, is restricted in certain respects from a linguistic point of view. One issue deserves to be singled-out: quantified sentences such as *Each woman gave at least two talks* abound in natural language. To provide an adequate semantic account of quantification might require an extension to

the form of the semantic translation rules, such as the ability to nest quantified expressions within one another. This would, in turn, necessitate a layer of interpretation between the semantic translations and the existing definition of a CSP in (2). An alternative approach might be to retain the direct correspondence between semantic translations and the specified form of a CSP, and instead enhance the underlying process of network consistency so that it is sensitive to different quantificational restrictions on semantic variables. Which of these, or other routes (such as Mellish (1985)), will be more profitable remains an open research topic, and one which will be influenced by future developments in both constraint-based reasoning and computational linguistics.

These issues are apparently orthogonal to the question of the connective structure of English semantics. Here there is a strong possibility that the tree-like form we have demonstrated of certain classes of semantic expression, represents a general tendency in the language as a whole. This line of thinking, together with Barton's earlier observations, suggests that the structures which have evolved in natural language may be a surface manifestation of an underlying reasoning system which is most effective with sparse, linearly connected problems. If this is true, then network consistency algorithms are a promising initial characterisation of that system.

Acknowledgements

I am grateful to Jon Oberlander, Mark Steedman, and Bonnie Webber for commenting on earlier incarnations of this work, and Kave Eshghi, Martin Sadler and Phil Stenton for more recent discussions. Any errors are of course my own.

References

Barton, G, E. (1986) Constraint Propagation in Kimmo Systems. In *Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics*, Columbia University, New York, N.Y., pp.45–52.

Barton, G. E., Berwick, R. C. and Ristad, E. S. (1987) *Computatational Complexity and Natural Language*. Cambridge, Mass.: MIT Press.

Crain, S. and Steedman, M. J. (1985) On Not Being Led up the Garden Path: The Use of Context by the Psychological Parser. In Dowty, D., Karttunen, L. and Zwicky, A. (eds.) *Natural Language Parsing: Psychological, Computational, and Theoretical Perspectives.*

Dale, R. and Haddock, N. J. (forthcoming) Generating Referring Expressions with Relations. To appear in *Proceedings of the Fifth Conference of the European Chapter of the Association for Computational Linguistics*, Berlin.

Dechter, R. and Pearl, J. (1988) Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence*, **34**, 1–38.

Dechter, R., Meiri, I., and Pearl, J. (1989) Temporal Constraint Networks. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada, pp.83–93.

Duffy, G. (1986) Categorial Disambiguation. In *Proceedings of the 5th Annual Meeting of the American Association for Artificial Intelligence*, Philadelphia, Pa., pp.1079–1082.

Engdahl, E. (1983) Parasitic Gaps. *Linguistics and Philosophy*, **6**, 5–34.

Freuder, E. C. (1978) Synthesising Constraint Expressions. *Communications of the ACM*, **21**, 958–966.

Freuder, E. C. (1982) A Sufficient Condition for Backtrack-Free Search. *Journal of the ACM*, **19**, 24–32.

Haddock, N. J. (1988) Incremental Semantics and Interactive Syntactic Processing. Unpub-

lished Ph.D. Thesis, Dept. of AI and Centre for Cognitive Science, University of Edinburgh.

Haddock, N. J. (1989) Computational Models of Incremental Semantic Interpretation. *Language and Cognitive Processes*, 4, 337–368.

Haddock, N. J. (1991) Linear-Time Reference Evaluation. Technical Report, Hewlett-Packard Laboratories, Bristol.

Mackworth, A. K. (1987) Constraint Satisfaction. In Shapiro, S. (ed) *The Encyclopedia of Artifical Intelligence*, Volume 1, pp.205–211. John Wiley and Sons.

Mackworth, A. K. (1977a) Consistency in Network of Relations. *Artificial Intelligence*, 8, 99–118.

Mackworth, A. K. (1977b) On Reading Sketch Maps. In *Proceedings of the Fifth International Joint Conference on Artifical Intelligence*, MIT, Cambridge, Mass., pp.598–606.

Maruyama, H. (1990) Structural Disambiguation with Constraint Propogation. In *Proceedings of the 28th Annual Meeting of the Association for Computational Linguistics*, pp.31–38.

Mellish, C.S. (1985) *Computer Interpretation of Natural Language Descriptions.* Chichester: Ellis Horwood.

Montanari, U. (1974) Networks of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Science*, 7, 95–132.

Partee, B. (1978) Bound Variables and Other Anaphors. In Waltz, D. L. (ed.) *Theoretical Issues in Natural Language Processing-2*, University of Illinois at Urbana-Champaign Urbana, Illinois, pp.79–85.

Pereira, F. C. N. and Shieber, S. M. (1987) *Prolog and Natural Language Analysis.* Center for the Study of Language and Information. CSLI Lecture Notes No. 10.

Pereira, F. C. N. and Pollack, M. E. (forthcoming) Incremental Interpretation. To appear in *Artificial Intelligence.*

Rich, E., Wittenburg, K., Barnett, J., Wroblewski, D. (1987) Ambiguity Procrastination. In *Proceedings of the 6th Annual Meeting of the American Association for Artificial Intelligence*, Seattle, pp.571–576.

Schubert, L. and Pelletier, J. (1982) From English to Logic: Context-Free Computation of 'Conventional' Logical Translation. *American Journal of Computational Linguistics*, 8, 27–44.

Winston, P. (1984) *Artificial Intelligence.* Reading, Mass.: Addison-Wesley.

Combining Qualitative and Quantitative Constraints in Temporal Reasoning*

Itay Meiri

Cognitive Systems Laboratory
Computer Science Department
University of California
Los Angeles, CA 90024
itay@cs.ucla.edu

January 30, 1991

Abstract

This paper presents a general model for temporal reasoning, capable of handling both qualitative and quantitative information. This model allows the representation and processing of all types of constraints considered in the literature so far, including metric constraints (restricting the distance between time points), and qualitative, disjunctive, constraints (specifying the relative position between temporal objects). Reasoning tasks in this unified framework are formulated as constraint satisfaction problems, and are solved by traditional constraint satisfaction techniques, such as backtracking and path consistency. A new class of tractable problems is characterized, involving qualitative networks augmented by quantitative domain constraints, some of which can be solved in polynomial time using arc and path consistency.

*This work was supported in part by the Air Force Office of Scientific Research, AFOSR 900136.

1 Introduction

In recent years, several constraint-based formalisms have been proposed for temporal reasoning, most notably, Allen's interval algebra (IA) [1], Vilain and Kautz's point algebra (PA) [14], Dean and McDermott's time map [3], and metric networks (Dechter, Meiri and Pearl [5]). In these formalisms, temporal reasoning tasks are formulated as constraint satisfaction problems, where the variables are temporal objects such as points and intervals, and temporal statements are viewed as constraints on the location of these objects along the time line. Unfortunately, none of the existing formalisms can conveniently handle all forms of temporal knowledge. Qualitative approaches such as Allen's interval algebra and Vilain and Kautz's point algebra face difficulties in representing and reasoning about metric, numerical information, while the quantitative approaches exhibit limited expressiveness when it comes to qualitative information [5].

In this paper we offer a general, network-based computational model for temporal reasoning, capable of handling both qualitative and quantitative information. In this model, variables represent both points and intervals (as opposed to existing formalisms, where one has to commit to a single type of objects), and constraints may be either metric, between points, or qualitative disjunctive relations between objects. The unique feature of this framework is that it allows the representation and processing of all types of constraints considered in the literature so far.

The main contribution of this paper lies in providing a formal unifying framework for temporal reasoning, generalizing the interval algebra, the point algebra, and metric networks. In this framework, we are able to utilize constraint satisfaction techniques in solving several reasoning tasks. Specifically:

1. General networks can be solved by decomposition into *singleton labelings*, each solvable in polynomial time. This decomposition scheme can be improved by traditional constraint satisfaction techniques such as variants of backtrack search.

2. The input can be effectively encoded in a *minimal network* representation, which provides answers to many queries.

3. Path consistency algorithms can be used in preprocessing the input network to improve search efficiency, or to compute an approximation to the minimal network.

4. We were able to identify two classes of tractable problems, solvable in polynomial time. The first consists of *augmented qualitative networks*, composed of qualitative constraints between points and quantitative domain constraints, which can be solved using arc and path consistency. The second class consists of networks for which path consistency algorithms are exact.

We also show that our model compares favorably with an alternative approach for combining quantitative and qualitative constraints, proposed by Ladkin [6], from both conceptual and computational points of view.

The paper is organized as follows. Section 2 formally defines the constraint types under consideration. The definitions of the new model are given in Section 3. Section 4 reviews

and extends the hierarchy of qualitative networks. Section 5 discusses *augmented qualitative networks*—qualitative networks augmented by domain constraints. Section 6 presents two methods for solving general networks: a decomposition scheme and path consistency, and identifies a class of networks for which path consistency is exact. Section 7 provides summary and concluding remarks, including a comparison to Ladkin's model. Proofs of theorems can be found in the extended version of this paper [10].

2 The Representation Language

Consider a typical temporal reasoning problem. We are given the following information.

> **Example 1.** John and Fred work for a company in LA. They usually work at the local office, in which case it takes John less than 20 minutes and Fred between 15–20 minutes to get to work. Twice a week John works at the main office, in which case he commutes at least 60 minutes to work. Today John left home between 7:05–7:10, and Fred arrived at work between 7:50–7:55. We also know that Fred and John met at a traffic light on their way to work.

We wish to represent and reason about such knowledge. We wish to answer queries such as: "is the information in this story consistent?," "who was the first to arrive at work?," "what are the possible times at which John arrived at work?," and so on.

We consider two types of temporal objects: points and intervals. Intervals correspond to time periods during which events occur or propositions hold, and points represent beginning and ending points of some events, as well as neutral points of time. For example, in our story, we have two meaningful events: "John was going to work" and "Fred was going to work." These events are associated with intervals $J = [P_1, P_2]$, and $F = [P_3, P_4]$, respectively. The extreme points of these intervals, $P_1, \ldots, P_4$, represent the times in which Fred and John left home and arrived at work. We also introduce a neutral point, P_0, to represent the "beginning of the world" in our story. One possible choice for P_0 is 7:00 a.m. Temporal statements in the story are treated as constraints on the location of objects (such as intervals J and F, and points $P_0, \ldots, P_4$) along the time line. There are two types of constraints: qualitative and quantitative. Qualitative constraints specify the relative position of pairs of objects. For instance, the fact that John and Fred met at a traffic light, forces intervals J and F to overlap. Quantitative constraints place absolute bounds or restrict the temporal distance between points. For example, the information on Fred's commuting time constrains the length of interval F, i.e., the distance between P_3 and P_4. In the rest of this section we formally define qualitative and quantitative constraints, and the relationships between them.

2.1 Qualitative Constraints

A qualitative constraint between two objects O_i and O_j, each may be a point or an interval, is a disjunction of the form

$$(O_i\ r_1\ O_j) \vee \cdots \vee (O_i\ r_k\ O_j), \tag{1}$$

where each one of the r_i's is a *basic relation* that may exist between the two objects. There are three types of basic relations.

- Basic *Interval-Interval (II) relations* that can hold between a pair of intervals [1], *before, meets, starts, during, finishes, overlaps*, their inverses, and the equality relation, a total of 13 relations, denoted by the set $\{b, m, s, d, f, o, bi, mi, si, di, fi, oi, =\}$.
- Basic *Point-Point (PP) relations* that can hold between a pair of points [14], denoted by the set $\{<, =, >\}$.
- Basic *Point-Interval (PI) relations* that can hold between a point and an interval, and basic *Interval-Point (IP) relations* that can hold between an interval and a point. These relations are defined in Table 1.

Relation	Symbol	Inverse	Relations on Endpoints
p before I	*b*	*bi*	$p < I^-$
p starts I	*s*	*si*	$p = I^-$
p during I	*d*	*di*	$I^- < p < I^+$
p finishes I	*f*	*fi*	$p = I^+$
p after I	*a*	*ai*	$p > I^+$

Table 1: The basic relations between a point p and an Interval $I = [I^-, I^+]$.

A subset of basic relations (of the same type) corresponds to an ambiguous, disjunctive, relationship between objects. For example, Equation (1) may also be written as $O_i \{r_1, \ldots, r_k\} O_j$; alternatively, we say that the constraint between O_i and O_j is the relation set $\{r_1, \ldots, r_k\}$. One qualitative constraint given in Example 1 reflects the fact that John and Fred met at a traffic light. It is expressed by an II relation specifying that intervals J and F are not disjoint:

$$J \{s, si, d, di, f, fi, o, oi, =\} F.$$

To facilitate the processing of qualitative constraints, we define a *qualitative algebra(QA)*, whose elements are all legal constraints (all subsets of basic relations of the same type)—2^{13} II Relations, 2^3 PP relations, 2^5 PI relations, and 2^5 IP relations. Two binary operations are defined on these elements: intersection and composition. The *intersection* of two qualitative constraints, R' and R'', denoted by $R' \oplus R''$, is the set-theoretic intersection $R' \cap R''$. The *composition* of two constraints, R' between objects O_i and O_j, and R'' between objects O_j and O_k, is a new relation between objects O_i and O_k, induced by R' and R''. Formally, the composition of R' and R'', denoted by $R' \otimes R''$, is the composition of the constituent basic relations, namely

$$R' \otimes R'' = \{r' \otimes r'' | r' \in R', r'' \in R''\}.$$

Composition of two basic relations r' and r'', is defined by a *transitivity table* shown in Table 2. Six transitivity tables, $T_1, \ldots, T_4, T_{PA}, T_{IA}$, are required; each defining a composition of basic relations of a certain type. For example, composition of a basic PP relation and a basic PI relation is defined in table T_1. Two important subsets of QA are Allen's Interval

Algebra (IA), the restriction of QA to II relations, and Vilain and Kautz's Point Algebra (PA), its restriction to PP relations. The corresponding transitivity tables are given in [1] and [14], and appear in Table 2 as T_{IA} and T_{PA}, respectively. The rest of the tables, $T_1, \ldots, T_4$, are given in the extended version of this paper [10]. Illegal combinations in Table 2 are denoted by $\emptyset$.

	PP	PI	IP	II
PP	$[T_{PA}]$	$[T_1]$	$[\emptyset]$	$[\emptyset]$
PI	$[\emptyset]$	$[\emptyset]$	$[T_2]$	$[T_4]$
IP	$[T_1]^t$	$[T_3]$	$[\emptyset]$	$[\emptyset]$
II	$[\emptyset]$	$[\emptyset]$	$[T_4]^t$	$[T_{IA}]$

Table 2: A full transitivity table.

2.2 Quantitative Constraints

Quantitative constraints refer to absolute location or the distance between *points* [5]. There are two types of quantitative constraints:

- A *unary* constraint, on point P_i, restricts the location of P_i to a given set of intervals
$$(P_i \in I_1) \vee \cdots \vee (P_i \in I_k).$$
- A *binary* constraint, between points P_i and P_j, constrains the permissible values for the distance $P_j - P_i$:
$$(P_j - P_i \in I_1) \vee \cdots \vee (P_j - P_i \in I_k).$$

In both cases the constraint is represented by a set of intervals $\{I_1, \ldots, I_k\}$; each interval may be open or closed in either side. For example, one binary constraint given in our story specifies the duration of interval J (the event "John was going to work)":

$$P_2 - P_1 \in \{(0, 20), (60, \infty)\}.$$

The fact that John left home between 7:05–7:10 is translated into a unary constraint on P_1, $P_1 \in \{(5, 10)\}$, or $5 < P_1 < 10$ (note that all times are relative to P_0, i.e. 7:00 a.m.). Sometimes it is easier to treat a unary constraint on P_i as a binary constraint between P_0 and P_i, having the same interval representation. For example, the above unary constraint is equivalent to the binary constraint, $P_1 - P_0 \in \{(5, 10)\}$.

The intersection and composition operations for quantitative constraints assume the following form. Let C' and C'' be quantitative constraints, represented by interval sets I' and I'', respectively. Then, the intersection of C' and C'' is defined as:

$$C' \oplus C'' = \{I_i \cap I_j | I_i \in I', I_j \in I''\}.$$

The composition of C' and C'' is a new interval set defined by:

$$C' \otimes C'' = \{z | \exists x \in I', y \in I'', \; x + y = z\}.$$

2.3 Relationships between Qualitative and Quantitative Constraints

The existence of a constraint of one type sometimes implies the existence of an implicit constraint of the other type. This can only occur when the constraint involves two points. Consider a pair of points P_i and P_j. If a quantitative constraint, C, between P_i and P_j is given (by an interval set $\{I_1, \ldots, I_k\}$), then the implied qualitative constraint, QUAL(C), is defined as follows (see also Ladkin [6]).

- If $0 \in \{I_1, \ldots, I_k\}$[1], then "=" $\in$ QUAL(C).
- If there exists a value $v > 0$ such that $v \in \{I_1, \ldots, I_k\}$, then "<" $\in$ QUAL(C).
- If there exists a value $v < 0$ such that $v \in \{I_1, \ldots, I_k\}$, then ">" $\in$ QUAL(C).

Similarly, If a qualitative constraint, C, between P_i and P_j is given (by a relation set R), then the implied quantitative constraint, QUAN(C), is defined as follows.

- If "<" $\in R$, then $(0, \infty) \in$ QUAN(C).
- If "=" $\in R$, then $[0] \in$ QUAN(C).
- If ">" $\in R$, then $(-\infty, 0) \in$ QUAN(C).

The intersection and composition operations can be extended to cases where the operands are constraints of different types. If C' is a quantitative constraint and C'' is qualitative, then intersection is defined as quantitative intersection:

$$C' \oplus C'' = C' \oplus \text{QUAN}(C''). \tag{2}$$

Composition, on the other hand, depends on the type of C''.

- If C'' is a PP relation, then composition (and consequently the resulting constraint) is quantitative
$$C' \otimes C'' = C' \otimes \text{QUAN}(C'').$$
- If C'' is a PI relation, then composition is qualitative
$$C' \otimes C'' = \text{QUAL}(C') \otimes C''.$$

3 General Temporal Constraint Networks

We now present a network-based model which facilitates the processing of all constraints described in the previous section. The definitions of the new model follow closely those developed for discrete constraint networks [11], and for metric networks [5].

[1] We use the convention that $v \in \{I_1, \ldots, I_k\}$ is equivalent to $v \in I_1$ or ... or $v \in I_k$

A *general temporal constraint network* involves a set of variables $\{X_1, \ldots, X_n\}$, each representing a temporal object (a point or an interval), and a set of unary and binary constraints. When a variable represents a time point its domain is the set of real numbers $\Re$; when a variable represents a temporal interval, its domain is the set of ordered pairs of real numbers, i.e. $\{(a,b)|a,b \in \Re, a < b\}$. Constraints may be quantitative or qualitative. Each qualitative constraint is represented by a relation set R. Each quantitative constraint is represented by an interval set I. Constraints between variables representing points are always maintained in their quantitative form. We also assume that unary quantitative constraints are represented by equivalent binary constraints, as shown in the previous section. A set of internal constraints relates each interval $I = [I^-, I^+]$ to its endpoints, I^- $\{starts\}$ I, and I^+ $\{finishes\}$ I.

A constraint network is associated with a *directed constraint graph*, where nodes represent variables, and an arc $i \rightarrow j$ indicates that a constraint C_{ij}, between variables X_i and X_j, is specified. The arc is labeled by an interval set (when the constraint is quantitative) or by a QA element (when it is qualitative). The constraint graph of Example 1 is shown in Figure 1.

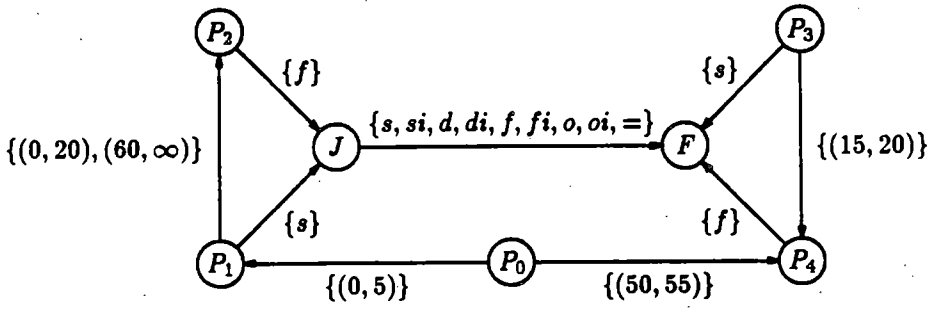


Figure 1: The constraint graph of Example 1.

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 (note that the value assigned to a variable which represents an interval is a pair of real numbers). The network is *consistent* if at least one solution exists. A value v is a *feasible value* for variable X_i, if there exists a solution in which $X_i = v$. The set of all feasible values of a variable is called its *minimal domain*.

We define a partial order, $\subseteq$, among binary constraints of the *same type*. A constraint C' is *tighter* than constraint C'', denoted by $C' \subseteq C''$, if every pair of values allowed by C' is also allowed by C''. If C' and C'' are qualitative, represented by relation sets R' and R'', respectively, then $C' \subseteq C''$ if and only if $R' \subseteq R''$. If C' and C'' are quantitative, represented by interval sets I' and I'', respectively, then $C' \subseteq C''$ if and only if for every value $v \in I'$, we have also $v \in I''$. This partial order can be extended to networks in the usual way. A network N' is tighter than network N'', if the partial order $\subseteq$ is satisfied for all the corresponding constraints. Two networks are *equivalent* if they possess the same solution set. A network may have many equivalent representations; in particular, there is a unique equivalent network, M, which is minimal with respect to $\subseteq$, called the *minimal network* (the minimal network is unique because equivalent networks are closed under intersection). The arc constraints specified by M are called the *minimal constraints*.

The minimal network is an effective, more explicit, encoding of the given knowledge. Consider for example the minimal network of Example 1. The minimal constraint between J and F is $\{di\}$, the minimal constraint between P_1 and P_2 is $\{(60, \infty)\}$, and the minimal constraint between P_0 and P_3 is $\{(30, 40)\}$. From this minimal network representation, we can infer that today John was working in the main office; he arrived at work after 8:00 a.m., while Fred arrived at work between 7:30–7:40.

Given a network N, the first interesting task is to determine its consistency. If the network is consistent, we are interested in other reasoning tasks, such as finding a solution to N, computing the minimal domain of a given variable X_i, computing the minimal constraint between a given pair of variables X_i and X_j, and computing the full minimal network. The rest of the paper is concerned with solving these tasks.

4 The Hierarchy of Qualitative Networks

Before we present solution techniques for general networks, we briefly describe the hierarchy of qualitative networks.

Consider a network having only qualitative constraints. If all constraints are II relations (namely IA elements), or PP relations (PA elements), then the network is called an *IA network*, or a *PA network*, respectively [12]. If all constraints are PI and IP relations, then the network is called an *IPA network* (for *Interval-Point* Algebra[2]). A special case of a PA network, where the relations are convex (taken only from $\{<, \leq, =, \geq, >\}$, namely excluding $\neq$), is called a *convex PA network* (*CPA network*).

It can be easily shown that any qualitative network can be represented by an IA network. On the other hand, there are some qualitative networks that cannot be represented by a PA network. For example (see [14]), a network consisting of two intervals, I and J, and a single constraint between them, $I\ \{before, after\}\ J$. Formally, the following relationship can be established among qualitative networks.

Proposition 1 *Let QN be the set of all qualitative networks. Let* $net(CPA)$, $net(PA)$, $net(IPA)$, *and* $net(IA)$ *denote the set of qualitative networks which can be represented by CPA networks, PA networks, IPA networks, and IA networks, respectively. Then,*

$$net(CPA) \subset net(PA) \subset net(IPA) \subset net(IA) = QN.$$

By climbing up in this hierarchy from CPA networks towards IA networks we gain expressiveness, but at the same time lose tractability. For example, deciding consistency of a PA network can be done in time $O(n^2)$ [13, 9], but it becomes NP-complete for IA networks [14], or even for IPA networks, as stated in the following theorem.

Theorem 2 *Deciding consistency of an IPA network is NP-hard.*

Theorem 2 suggests that the border between tractable and intractable qualitative networks lies somewhere between PA networks and IPA networks.

[2]We use this name to comply with the names IA and PA, although technically these relations, together with the intersection and composition operations, do not constitute an algebra, because they are not closed under composition.

5 Augmented Qualitative Networks

We now return to solving general networks. First, we observe that even the simplest task of deciding consistency of a general network is NP-hard. This follows trivially from the fact that deciding consistency for either metric networks or IA networks is NP-hard [2, 5, 14]. Therefore, it is unlikely that there exists a general polynomial algorithm for deciding consistency of a network. In this section we take another approach, and pursue "islands of tractability"—special classes of networks which admit polynomial solution. In particular, we consider the simplest type of network which contains both qualitative and quantitative constraints, called an *augmented qualitative network*, a qualitative network augmented by unray constraints on its domains.

We may view qualitative networks as a special case of augmented qualitative networks, where the domains are unlimited. For example, PA networks can be regarded as augmented qualitative networks with domains $(-\infty, \infty)$. It follows that in our quest for tractability, we can only augment tractable qualitative networks such as CPA and PA networks.

In this section, we consider CPA and PA networks over three domain classes which carry significant importance in temporal reasoning applications:

1. *Discrete domains*, where each variable may assume only a finite number of values (for instance, when we settle for crude timing of events such as the day or year in which they occurred).

2. *Single-interval domains*, where we have only an upper and/or a lower bound on the timing of events.

3. *Multiple-intervals domains*, which subsumes the two previous cases[3].

A CPA network over multiple-intervals domains is depicted in Figure 2a, where each variable is labeled by its domain intervals. Note that in this example, and also throughout the rest of this section, we express the domain constraints as unary constraints.

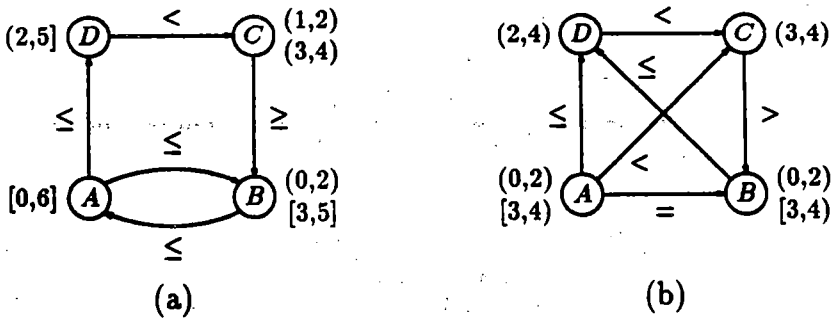


Figure 2: (a) A CPA network over multiple-intervals domains. (b) An equivalent arc-consistent and path-consistent form.

We next show that for augmented CPA networks and for some augmented PA networks, all interesting reasoning tasks can be solved in polynomial time, by enforcing arc consistency (AC) and path consistency (PC).

[3]Note that a discrete domain $\{v_1, \ldots, v_k\}$ is essentially a multiple-intervals domain $\{[v_1, v_1], \ldots, [v_k, v_k]\}$.

First, let us review the definitions of arc consistency and path consistency [7, 11]. An arc $i \rightarrow j$ is *arc consistent* if and only if for any value $x \in D_i$, there is a value $y \in D_j$, such that the constraint C_{ij} is satisfied. A path P from i to j, $i_0 = i \rightarrow i_1 \rightarrow \cdots \rightarrow i_m = j$, is *path consistent* if the direct constraint C_{ij} is tighter than the composition of the constraints along P, namely, $C_{ij} \subseteq C_{i_0,i_1} \otimes \cdots \otimes C_{i_{m-1},i_m}$. Note that our definition of path consistency is slightly different than the original one [7], as it does not consider the domain constraints. A *network* G is arc (path) consistent if all its arcs (paths) are consistent. Figure 2b shows an equivalent arc- and path-consistent form of the network in Figure 2a.

The following theorems establish the local consistency levels which are sufficient to determine the consistency of augmented CPA networks.

Theorem 3 *Any nonempty*[4] *arc-consistent CPA network over discrete domains is consistent.*

Theorem 4 *Any nonempty arc- and path-consistent CPA network over multiple-intervals domains is consistent.*

Theorems 3 and 4 provide an effective test for deciding consistency of an augmented CPA network. We simply enforce the required consistency level, and then check whether the domains are empty. The network is consistent if and only if all the domains are nonempty. Moreover, by enforcing arc and path consistency we also compute the minimal domains, as stated in the next theorem.

Theorem 5 *Let $G = (V, E)$ be a nonempty arc- and path-consistent CPA network over multiple-intervals domains. Then, all domains are minimal.*

When we move up in the hierarchy from CPA networks to PA networks (allowing also the inequality relation between points), deciding consistency and computing the minimal domains remain tractable for single-interval domains. Unfortunately, deciding consistency becomes NP-hard for discrete domains, and consequently, for multiple-intervals domains.

Theorem 6 *Let $G = (V, E)$ be a nonempty arc- and path-consistent PA network over single-interval domains. Then, G is consistent, and all domains are minimal.*

Proposition 7 *Deciding consistency of a PA network over discrete domains is NP-hard.*

One way to convert a network into an equivalent arc-consistent form is by applying algorithm AC-3 [7], shown in Figure 3. The algorithm repeatedly applies the function REVISE$((i,j))$, which makes arc $i \rightarrow j$ consistent, until a fixed point, where all arcs are consistent, is reached. The function REVISE$((i,j))$ restricts D_i, the domain of variable X_i, using operations on quantitative constraints:

$$D_i \leftarrow D_i \oplus D_j \otimes \mathrm{QUAN}(C_{ji}).$$

Taking advantage of the special features of PA networks, we are able to bound the running time of AC-3 as follows.

[4] A nonempty network is a network in which all domains are nonempty.

1. $Q \leftarrow \{i \rightarrow j | i \rightarrow j \in E\}$
2. **while** $Q \neq \emptyset$ **do**
3. select and delete any arc $k \rightarrow m$ from Q
4. **if** REVISE((k, m)) **then**
5. $Q \leftarrow Q \cup \{i \rightarrow k | i \rightarrow k \in E, i \neq m\}$
6. **end**

Figure 3: AC-3—an arc consistency algorithm.

1. $Q \leftarrow \{(i, k, j) | (i < j), (k \neq i, j)\}$
2. **while** $Q \neq \emptyset$ **do**
3. select and delete any triplet (i, k, j) from Q
4. **if** REVISE((i, k, j)) **then**
5. $Q \leftarrow Q \cup$ RELATED-PATHS((i, k, j))
6. **end**

Figure 4: PC-2—a path consistency algorithm.

Theorem 8 *Let $G = (V, E)$ be an augmented PA network. Let n and e be the number of nodes and the number of edges, respectively. Then, the timing of algorithm AC-3 is bounded as follows.*

- *If the domains are discrete, then AC-3 takes $O(ek \log k)$ time, where k is the maximum domain size.*
- *If the domains consist of single intervals, then AC-3 takes $O(en)$ time.*
- *If the domains consist of multiple intervals, then AC-3 takes $O(enK \log K)$ time, where K is the maximum number of intervals in any domain.*

A network can be converted into an equivalent path-consistent form by applying any path consistency algorithm to the underlying qualitative network [7, 14, 12]. Path consistency algorithms impose local consistency among triplets of variables, (i, k, j), by using a relaxation operation

$$C_{ij} \leftarrow C_{ij} \oplus C_{ik} \otimes C_{kj}. \tag{3}$$

Relaxation operations are applied until a fixed point is reached, or until some constraint becomes empty indicating an inconsistent network. One efficient path consistency algorithm is PC-2 [7], shown in Figure 4, where the relaxation operation of Equation (3) is performed by the function REVISE((i, k, j)). Algorithm PC-2 runs to completion in $O(n^3)$ time [8].

Table 3 summarizes the complexity of determining consistency in augmented qualitative networks. Note that when both arc and path consistency are required, we first need to establish path consistency, which results in a complete graph, namely $e = n^2$. Algorithms

for assembling a solution to augmented qualitative networks are given in the extended version of this paper [10]. Their complexity is bounded by the time needed to decide consistency.

	Discrete	Single interval	Multiple intervals
CPA networks	AC $O(ek \log k)$	AC + PC $O(n^3)$	AC + PC $O(n^3 K \log K)$
PA networks	NP-complete	AC + PC $O(n^3)$	NP-complete
IPA networks	NP-complete	NP-complete	NP-complete

Table 3: Complexity of deciding consistency in augmented qualitative networks.

6 Solving General Networks

In this section we focus on solving general networks. First, we describe an exponential brute-force algorithm, and then we investigate the applicability of path consistency algorithms.

Let N be a given network. A *basic label* of arc $i \rightarrow j$, is a selection of a single interval from the interval set (if C_{ij} is quantitative) or a basic relation from the QA element (if C_{ij} is qualitative). A network whose arcs are labeled by basic labels of N is called a *singleton labeling* of N. We may solve N by generating all its singleton labelings, solve each one of them independently, and then combine the results. Specifically, N is consistent if and only if there exists a consistent singleton labeling of N, and the minimal network can be computed by taking the union over the minimal networks of all the singleton labelings.

Each qualitative constraint in a singleton labeling can be translated into a set of up to four linear inequalities on points. For example, a constraint I $\{during\}$ J, can be translated into linear inequalities on the endpoints of I and J, $I^- > J^-$, $I^- < J^+$, $I^+ > J^-$, and $I^+ < J^+$. Using the QUAN translation, these inequalities can be translated into quantitative constraints. It follows, that a singleton labeling is equivalent to an *STP network*—a metric network whose constraints are labeled by single intervals [5]. An STP network can be solved in $O(n^3)$ time [5]; thus, the overall complexity of this decomposition scheme is $O(n^3 k^e)$, where n is the number of variables, e is the number of arcs in the constraint graph, and k is the maximum number of basic labels on any arc.

This brute-force enumeration can be pruned significantly by running a backtracking algorithm on a meta-CSP whose variables are the network arcs, and their domains are the possible basic labels. Backtrack assigns a basic label to an arc, as long as the corresponding STP network is consistent and, if no such assignment is possible, it backtracks.

Imposing local consistency among subsets of variables may serve as a preprocessing step to improve backtrack. This strategy has been proven successful (see [4]), as enforcing local consistency can be achieved in polynomial time, while it may substantially reduce the number of dead-ends encountered in the search phase itself. In particular, experimental evaluation shows that enforcing a low consistency level, such as arc or path consistency, gives the best results [4]. Following this rationale, we next show that path consistency, which in general

networks amounts to the least amount of preprocessing, can be enforced in polynomial time.

To assess the complexity of PC-2 in the context of general networks, we introduce the notion of a *range* of a network [5]. The *range* of a quantitative constraint C, represented by an interval set $\{I_1, \ldots, I_k\}$, is $\sup(I_k) - \inf(I_1)$. The range of a *network* is the maximum range over all its quantitative constraints. The next theorem shows that the timing of PC-2 is bounded by $O(n^3R^3)$, where R is the range of the network (expressed in terms of the coarsest possible time units).

Theorem 9 *Let $G = (V, E)$ be a given network. Algorithm PC-2 performs no more than $O(n^3R)$ relaxation steps, and its timing is bounded by $O(n^3R^3)$, where R is the range of G.*

Path consistency can also be regarded as an alternative approach to exhaustive enumeration, serving as an approximation scheme which often yields the minimal network. For example, applying path consistency to the network of Figure 1 produces the minimal network. Although, in general, a path consistent network is not necessarily minimal, in some cases, path consistency is guaranteed to compute the minimal network, as stated in the following theorem.

Theorem 10 *Let $G = (V, E)$ be a path-consistent network. If the qualitative subnetwork of G is in net(CPA), and the quantitative subnetwork constitutes an STP network, then G is minimal.*

Corollary 11 *Any path-consistent singleton labeling is minimal.*

We feel that some more temporal problems can be solved by path consistency algorithms; further investigation may reveal new classes for which these algorithms are exact.

7 Conclusions

We described a general network-based model for temporal reasoning capable of handling both qualitative and quantitative information. It facilitates the processing of quantitative constraints on points, and all qualitative constraints between temporal objects. We used constraints satisfaction techniques in solving reasoning tasks in this model. In particular, general networks can be solved by a backtracking algorithm, or by path consistency, which computes an approximation to the minimal network.

Ladkin [6] has introduced an alternative model for temporal reasoning. It consists of two components: a metric network and an IA network. These two networks, however, are not connected via internal constraints, rather, they are kept separately, and the inter-component relationships are managed by means of external control. To solve reasoning tasks in this model, Ladkin proposed an algorithm which solves each component independently, and then circulates information between the two parts, using the QUAL and QUAN translations, until a fixed point is reached. Our model has two advantages over Ladkin's model:

1. It is conceptually clearer, as all information is stored in a single network, and constraint propagation takes place in the knowledge level itself.

2. From computational point of view, it saves much of the redundant work done by circulating information between the two components. For example, in order to convert a given network into an equivalent path-consistent form, Ladkin's algorithm may require $O(n^2)$ informations transferences, resulting in an overall complexity of $O(n^5R^3)$, compared to $O(n^3R^3)$ in our model.

Using our integrated model we were able to identify two new classes of tractable networks. The first class is obtained by augmenting PA and CPA networks with various domain constraints. We showed that some of these networks can be solved using arc and path consistency. The second class consists of networks which can be solved by path consistency algorithms, for example, singleton labelings.

Future research should enrich the representation language to facilitate modeling of more involved reasoning tasks; in particular, we should incorporate non-binary constraints in our model.

Acknowledgments

I would like to thank Rina Dechter and Judea Pearl for providing helpful comments on an earlier draft of this paper.

References

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

[2] E. Davis. private communication.

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

[4] R. Dechter and I. Meiri. Experimental evaluation of preprocessing techniques in constraint satisfaction problems. In *Proceedings of IJCAI-89*, pages 271–277, Detroit, 1989.

[5] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, to appear, 1991.

[6] P. B. Ladkin. Metric constraint satisfaction with intervals. Technical Report TR-89-038, International Computer Science Institute, Berkeley, CA, 1989.

[7] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[8] A. K. Mackworth and E. C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems. *Artificial Intelligence*, 25(1):65–74, 1985.

[9] I. Meiri. Faster constraint satisfaction algorithms for temporal reasoning. Technical Report TR-151, UCLA Cognitive Systems Laboratory, Los Angeles, CA, 1990.

[10] I. Meiri. Combining qualitative and quantitative constraints in temporal reasoning. Technical Report TR-160, UCLA Cognitive Systems Laboratory, Los Angeles, CA, 1991.

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

[12] P. van Beek. ***Exact and Approximate Reasoning about Qualitative Temporal Relations.*** PhD thesis, University of Waterloo, Ontario, Canada, 1990.

[13] P. van Beek. Reasoning about qualitative temporal information. In *Proceedings of AAAI-90*, pages 728–734, Boston, 1990.

[14] M. Vilain and H. Kautz. Constraint propagation algorithms for temporal reasoning. In *Proceedings of AAAI-86*, pages 377–382, 1986.

Integrating Metric and Qualitative Temporal Reasoning

Henry A. Kautz
AT&T Bell Laboratories
Murray Hill, NJ 07974
kautz@research.att.com

Peter B. Ladkin
International Computer Science Institute
Berkeley, CA 94704
ladkin@icsi.berkeley.edu

abstract

Research in Artificial Intelligence on constraint-based representations for temporal reasoning has largely concentrated on two kinds of formalisms: systems of simple linear inequalities to encode metric relations between time points, and systems of binary constraints in Allen's temporal calculus to encode qualitative relations between time intervals. Each formalism has certain advantages. Linear inequalities can represent dates, durations, and other quantitive information; Allen's qualitative calculus can express relations between time intervals, such as disjointedness, that are useful for constraint-based approaches to planning.

In this paper we demonstrate how metric and Allen-style constraint networks be integrated in a constraint-based reasoning system. The highlights of the work include a simple but powerful logical language for expressing both quantitative and qualitative information; translation algorithms between the metric and Allen sublanguages that entail minimal loss of information; and a constraint-propagation procedure for problems expressed in a combination of metric and Allen constraints.

DRAFT

January 31, 1991

1 Introduction

Research in Artificial Intelligence on constraint-based representations for temporal reasoning has largely concentrated on two kinds of formalisms: systems of simple linear inequalities [Malik and T.O., 1983, Valdes-Perez, 1986, Dechter *et al.*, 1989] to encode metric relations between time points, and systems of binary constraints in Allen's temporal calculus [Allen, 1983, Vilain *et al.*, 1989, Ladkin and Madux, 1987, van Beek and Cohen, 1989] to encode qualitative relations between time intervals. Each formalism has certain advantages. Linear inequalities can represent dates, durations, and other quantitive information that appears in real-world planning and scheduling problems. Allen's qualitative calculus can express certain crucial relations between time intervals, such as *disjointedness*, that *cannot* be expressed by any collection of simple linear inequalities (without specifying which interval is before the other). Such disjointedness constraints form the basis for constraint-based approaches to planning [Allen, 1991].

In this paper we demonstrate how metric and qualitative knowledge can be integrated in a constraint-based reasoning system. One approach to this problem (as used, for example, in the "time map" system of Dean and McDermott [87]) is to directly attach rules that enforce disjointedness constraints to a network of linear inequalities. One limitation of such an approach is that some natural qualitative inferences are not performed: for example, the facts that interval i is during j and j is disjoint from k are not combined to reach the conclusion that i is disjoint from k. Another disadvantage is that it is often more convenient for the user to enter assertions in a qualitative language, even if they can be represented numerically.

Instead of try to augment a single reasoning system, we will take an approach briefly suggested by Dechter, Meiri, and Pearl [89] (henceforth "DMP"), and combine a metric reasoning system with a full Allen-style constraint network. The contributions of our research include the following:

1. A simple but powerful logical language $\mathcal{L}$ for expressing both quantitative and qualitative information. The language subsumes both networks of two-variable difference inequalities (called $\mathcal{L}_M$) and networks of binary Allen constraints (called $\mathcal{L}_A$), but is much more powerful than either. The axioms of Allen's temporal calculus are *theorems* of $\mathcal{L}$.

2. An extension of DMP's algorithms for networks of non-strict inequalities to handle both the strict and the non-strict inequalities that appear in $\mathcal{L}_M$.

3. Optimal translations between $\mathcal{L}_M$ and $\mathcal{L}_A$. As we noted, the two formalisms have orthogonal expressive power, so an exact translation is impossible; we say that a translation is *optimal* when it entails a minimal loss of information. Formally, $f : \mathcal{L}_1 \rightarrow \mathcal{L}_2$ is optimal iff for any $\alpha \in \mathcal{L}_1$ and $\beta \in \mathcal{L}_2$, then $\alpha \models \beta$ iff $f(\alpha) \models \beta$.

4. A constraint-propagation procedure for the combined constraint language $\mathcal{L}_M \cup \mathcal{L}_A$, which is based on the translation algorithms. The user of the system is able to enter information in terms of point difference inequalities or qualitative interval constraints, whichever is necessary or most convenient.

The system we describe in this paper is completely implemented in Common Lisp, and is available from the first author.

2 A Universal Temporal Language

Consider the following model of time: time is linear, and time points can be identified with the rationals under the usual ordering $<$. The difference of any two time points is likewise a rational number. An interval is a pair of points $\langle n, m \rangle$, where $n < m$. Two intervals stand in a particular qualitative relationship such as "overlaps" just when their endpoints stand in a particular configuration — in this case, when the starting point of the first falls before the starting point of the second, and the final point of the first falls between the two points of the second.

The following language $\mathcal{L}$ lets us say everything we'd like to about this model. It is typed predicate calculus with the following types and symbols:

types are Rational and Interval.
functions are
L, R : Interval $\Longrightarrow$ Rational
Intuitively, i_L is the starting (left) endpoint of i,
and i_R is the final (right) endpoint.
$-$ (subtraction): Rational $\times$ Rational $\Longrightarrow$Rational
Functions to construct rational numerals. (We use ordinary decimal notation in this paper.)
Rational numerals.
predicates are
$<, \leq, =$: Rational $\times$ Rational
Allen Predicates : Interval $\times$ Interval

P(recedes), M(eets), O(verlaps), S(tarts), D(uring),
F(inishes), =, and the inverses $P^{\smile}$, $M^{\smile}$, $O^{\smile}$, $S^{\smile}$, $D^{\smile}$, $F^{\smile}$.

The language does not include constants to name specific intervals; instead, we use unbound variables to name intervals, with the understanding that any particular model provides an interpretation for free variables.

It is useful to distinguish two special syntactic forms. Formulas of the form

$$i(r_1)j \vee \cdots \vee i(r_n)j$$

where the i and j are intervals and the r_i are Allen predicates are called *simple Allen constraints*, and are abbreviated as

$$i(r_1 + \cdots + r_n)j$$

The sublanguage of such formulas is called $\mathcal{L}_A$. Formulas of the following two forms

$$i_F - j_G < n \qquad i_F - j_G \leq n$$

where $F, G \in \{L, R\}$ and n is a numeral are called *simple metric constraints.* The sublanguage of such formulas is called $\mathcal{L}_M$. Note, however, that $\mathcal{L}$ is much richer than the union of $\mathcal{L}_M$ and $\mathcal{L}_A$. For example, the formulas in Table 1 are part of $\mathcal{L}$, but appear in neither $\mathcal{L}_A$ nor $\mathcal{L}_M$.

The following axioms capture the intended model of time.

- Arithmetic axioms for $-$ (subtraction), $<$, $\leq$, and numerals.
- $\forall i \,.\, i_L < i_R$
- Meaning postulates for each Allen predicate. The axioms for the non-inverted predicates appear in Table 1.

We write $C \models_{\mathcal{L}} D$ to mean that D holds in all of models of C that satisfy these axioms.

The original presentation of the Allen calculus described the predicates by a set of *transitivity axioms* such as

$$\forall i, j, k \,.\, i(M)j \wedge j(D)k \supset i(D + S + O)k$$

All of these formulas are *theorems* of $\mathcal{L}$, rather than axioms [Kautz and Ladkin, 1991].

$\forall i,j\,.$	$i=j$	$\equiv$	$i_L - j_L \leq 0$	$\wedge$	$j_L - i_L \leq 0$	$\wedge$	$i_R - j_R \leq 0$
						$\wedge$	$j_R - i_R \leq 0$
$\forall i,j\,.$	$i(P)j$	$\equiv$	$i_R - j_L < 0$				
$\forall i,j\,.$	$i(M)j$	$\equiv$	$i_R - j_L \leq 0$	$\wedge$	$j_l - i_R \leq 0$		
$\forall i,j\,.$	$i(O)j$	$\equiv$	$i_L - j_L < 0$	$\wedge$	$j_L - i_R < 0$	$\wedge$	$i_R - j_R < 0$
$\forall i,j\,.$	$i(S)j$	$\equiv$	$i_L - j_L \leq 0$	$\wedge$	$j_L - i_L \leq 0$	$\wedge$	$i_R - j_R < 0$
$\forall i,j\,.$	$i(D)j$	$\equiv$	$j_L - i_L < 0$	$\wedge$	$i_R - j_R < 0$		
$\forall i,j\,.$	$i(F)j$	$\equiv$	$j_L - i_L < 0$	$\wedge$	$i_R - j_R \leq 0$	$\wedge$	$j_R - i_R \leq 0$

Table 1: Meaning postulates for Allen predicates.

Since $\mathcal{L}$ is just first-order logic, we could solve problems that involve both metric and Allen assertions by employing a complete and general inference method, such as resolution. This is almost certain to be impractically slow. On the other hand, it appears that we do not need the full power of $\mathcal{L}$ to express many interesting temporal reasoning problems. The sublanguage $\mathcal{L}_M$ can express constraints on the duration of an interval (e.g., $i_L - i_R < -3$); on the elapsed time between intervals (e.g., $i_R - j_L < 5$); and between an interval and an absolute date, which we handle by introducing a "dummy" interval which is taken to begin at time number 0 (e.g., $i_L - day0_L < -14$). But $\mathcal{L}_M$ by itself is not adequate for many problems. For example, in the sublanguage $\mathcal{L}_A$ one can assert that intervals i and j are disjoint by the formula $i(P + M + M^{\smile} + P^{\smile})j$, but there is no equivalent formula in $\mathcal{L}_M$. Such a disjointedness constraint is useful in planning; for example, if i is a time during which a robot holds a block, and j is a time during which the robot's hand is empty, a planning system might want to make the assertion that $i(P+M+M^{\smile}+P^{\smile})j$. Another useful expression in $\mathcal{L}_A$ is $i(S+F)j$, which means that interval i starts or finishes j; for example, in scheduling a conference, you might want to assert that a certain talk begins or ends the conference.

So $\mathcal{L}_M \cup \mathcal{L}_A$ appears to be a good candidate for a practical temporal language. In order to develop an inference procedure for this language, let us examine the constraint satisfaction procedures that are known for $\mathcal{L}_M$ and $\mathcal{L}_A$ individually.

3 Constraint Networks

$\mathcal{L}_M$ and $\mathcal{L}_A$ can each express certain *binary constraint satisfaction problems* (CSP) [Montanari, 74]. A binary CSP is simply a set (also called a *network*)

of quantifier-free assertions in some language, each containing two variables. One possible task is to find a particular assignment of values to the variables that simultaneously satisfies all the constraints in the network: that is, to find a *model* of the network. (Henceforth in this paper we will always talk in terms of models rather than variable assignments.) Another important task is to compute the *minimal network representation* of the problem, which is defined as follows:

Definition: Minimal Network Representation
Suppose G is a consistent network of binary constraints in some language. Then a binary constraint network G' in that language is a minimal network representation of G iff the following all hold:

1. G' is logically equivalent to G.
2. For every pair of variables in G there is a constraint containing those variables in G'.
3. For any model $\mathcal{M}$ of a single constraint in G', there is a model $\mathcal{M}'$ for all of G' which agrees with $\mathcal{M}$ on the interpretation of the variables that appear in that constraint.

Hence from the minimal network representation one can "read off" the possible values that can be assigned to any variable.

$\mathcal{L}_M$ is very similar to what DMP called *simple temporal constraint satisfaction problems* (STCSP). They considered sets (or *networks*) of formulas of the form

$$n \leq (x - y) \leq m$$

where x and y are variables and n and m are numbers. Their representation differs from $\mathcal{L}_M$ in the following ways: (1) They abbreviated two inequalities in one formula, which is, of course, unimportant. (2) They use simple variables like x for time points, where $\mathcal{L}_M$ uses terms like i_L and i_R. Again this difference is not significant, because the interpretation of an interval i is simply the pair consisting of the interpretations of i_L and i_R. So we can treat i_L and i_R as "variables" in the CSP formulation. (3) Formulas in $\mathcal{L}_M$ include strict ($<$) as well as non-strict ($\leq$) inequalities.

DMP proved that an all-pairs shortest-path algorithm [Aho *et al.*, 1976, page 198] can compute the minimal network representation of a STCSP. One can modify the algorithm to handle the two kinds of inequalities as follows. We represent a formula M from $\mathcal{L}_M$ by a graph, where the nodes are the

terms that appear in M (that is, i_L and i_R for each interval variable i), and the directed arc from i_F to j_G is labeled with the pair $\langle n, 1\rangle$ if

$$i_F - j_G \leq n$$

appears in M, and labeled $\langle n, 0\rangle$ if

$$i_F - j_G < n$$

appears in M. Next we add the constraints from $\mathcal{L}$ that state that the left point of an interval is before its right point; that is, we add an arc $i_L\langle 0, 0\rangle i_R$ for each i. Finally we compute the shortest distance between all nodes in the graph using the following definitions for comparison and addition:

$$\langle m, x\rangle < \langle n, y\rangle \equiv m < n \vee (m = n \wedge x < y)$$

$$\langle m, x\rangle + \langle n, y\rangle = \langle m + n, \min(x, y)\rangle$$

The result is the minimal network representation of M. An arc appears between every pair of nodes in the graph, and the inequalities corresponding to the arcs are the strongest such inequalities implied by M. This procedure takes $O(n^3)$ time.

Binary CSP's based on the qualitative language $\mathcal{L}_A$ have been studied extensively [Allen, 1983, Ladkin and Madux, 1987, Vilain *et al.*, 1989, van Beek and Cohen, 1989]. Computing the minimal network representation of a set of such constraints is NP-Hard. In practice, however, one can approximate the minimal network by a weaker notion, called *n-consistency.* While we do not have space here to discuss the details of n-consistency, we note that the original presentation of $\mathcal{L}_A$ by Allen [83] included an algorithm that computes "3-consistency" in $O(n^3)$ time, and VanBeek [89] studied the improvements to the approximation likely to be found by computing higher degrees of consistency. For any *fixed* n, n-consistency can be computed in polynomial time.

Thus we have an efficient and complete algorithm for inference in $\mathcal{L}_M$, and a number of efficient approximation algorithms for $\mathcal{L}_A$. Figure 3 presents a constraint satisfaction algorithm for the union of the two languages. The method is to separately compute the minimal network representation of the metric and Allen constraints; derive new Allen constraints from the metric network and add these to the Allen network; derive new metric constraints from the Allen network and add these to the metric network; and repeat this process until no new statements can be derived. The system answers any query in $\mathcal{L}_M \cup \mathcal{L}_A$ by examining the appropriate network. The procedure is clearly correct; but now we must see how to translate $\mathcal{L}_M$ to $\mathcal{L}_A$ and vice-versa.

function combined-metric-Allen(M, A) =
input: simple metric network M and simple Allen network A
output: minimal networks M', A' implied by $M \cup A$

repeat
$A' :=$ **metric-to-Allen**$(M) \cup A$
$M' :=$ **Allen-to-metric**$(A') \cup M$
$M := M'$; $A := A'$
until $A = A'$ and $M = M'$
return M', A'
end combined-metric-Allen

Figure 1: Inference procedure for $\mathcal{L}_M \cup \mathcal{L}_A$.

4 Translating and Combining Metric and Allen Constraints

This section presents the optimal translations between the metric and Allen constraint languages, and a complexity analysis of the combined inference algorithm. We begin with the translation from $\mathcal{L}_M$ to $\mathcal{L}_A$. At first impression, one might think that it is sufficient to convert each metric constraint to the Allen constraint it implies. For example, from the meaning postulates one can deduce that

$$i_L - j_L < 0 \supset i(P + M + O + F^{\smile} + D^{\smile})j$$

So, if the metric network M contains $i_L - j_L < -3$ (which implies the antecedent of the formula), the translation includes $i(P + M + O + F^{\smile} + D^{\smile})j$. This approach is correct, but fails to capture all implications in $\mathcal{L}_M$. For example, suppose M is the following network:

$$\begin{aligned} i_L - i_R &< -3 \\ j_R - j_L &< 2 \end{aligned}$$

The minimal network representation of M has only trivial constraints between i and j (such as $i_L - j_R < \infty$), so the approach just outlined fails to infer that i *cannot* be during j, because i has longer duration than j.

Therefore an optimal translation must consider several metric constraints at a time; but how many? One might imagine that the problem required an exponential procedure that checked consistency of every possible Allen constraint between two intervals with all of M. Fortunately, this is not necessary:

function metric-to-Allen(M) =
input: a simple metric constraint network M.
output: the strongest set of simple Allen constraints implied by M.

```
    let M' be the minimal network representation of M
    if M' is inconsistent then return any inconsistent Allen network
    A_M := ∅
    for each pair of intervals i, j do
        let S be the {i_L, i_R, j_L, j_R} subnet of M'
        R := ∅
        for each primitive Allen relation r do
            S' := S ∪ { m | m is a simple metric constraint
                            in the meaning postulate for i(r)j }
            if S' is consistent then R := R ∪ {r}
        end do
        A_M := A_M ∪ {i(R)j}
    enddo
return A_M
end metric-to-Allen
```

Figure 2: Converting simple metric constraints to simple Allen constraints.

we can compute the strongest set of implied Allen constraints by considering constraints between just four points (that is, two intervals) at a time. The algorithm **metric-to-Allen** appears in Figure 2, and the following theorem formally states that it is optimal.

Theorem 1 *The algorithm* **metric-to-Allen** *is correct and entails minimal loss of information: For any $M \in L_M$ and $A \in L_A$, it's the case that $M \models_{\mathcal{L}} A$ iff* **metric-to-Allen**$(M) \models_{\mathcal{L}} A$. *The algorithm runs in $O(n^2)$ time, where n is the number of intervals.*

Proof: By theorem 2 of [Dechter *et al.*, 1989], any consistent and minimal simple metric network is decomposable. This means that any assignment of values to a set of terms that satisfies the subnet containing those terms can be extended to a satisfying assignment for the entire net. Another way of saying this is that if such a subnet has a model, then the net has a model that agrees with the subnet's model on the interpretation of the terms in the subnet.

Note that if two models agree on the interpretations of the terms i_L, i_R, j_L, j_R then they assign the same truth value to the expression $i(r)j$ where r is any primitive Allen

relation. From the construction of S' it is therefore the case that S' is consistent iff S' has a model iff S has a model in which $i(r)j$ holds iff M' has a model in which $i(r)j$ holds. Since M and M' are logically equivalent, we see that for any pair of intervals i and j, $i(R)j \in$ **metric-to-Allen**(M) iff for all $r \in R$ and no $r \notin R$, M has some model in which $i(r)j$ holds.

To show that the algorithm is correct, suppose that $i(R)j \in$ **metric-to-Allen**(M). If this clause were not implied by M, then there would be some model of M in which i and j stand in an Allen relation not in R. But that is impossible, as stated above. So $M \models_{\mathcal{L}}$**metric-to-Allen**(M), and **metric-to-Allen**$(M) \models_{\mathcal{L}} A$ implies $M \models_{\mathcal{L}} A$.

To show that the algorithm entails minimal loss of information, suppose that $M \models_{\mathcal{L}} A$. Because A is a conjunction of statements of the form $i(R)j$, we can assume without loss of generality that it is a single such statement. From the operation of the algorithm we see that there is some R' such that $i(R')j \in$ **metric-to-Allen**(M). We claim that $R' \subset R$. Suppose not; then there would be an $r \in R'$ such that $r \notin R$. But the former means that there is a model of M in which $i(r)j$ holds, and the latter means that there is no such model, since in any particular model only a single Allen relation holds between a pair of intervals. So since $R' \subset R$ means that $i(R')j$ implies $i(R)j$, it follows that **metric-to-Allen**$(M) \models_{\mathcal{L}} i(R)j$.

The complexity $O(n^2)$ follows immediately from the iteration of the outer loop; everything inside takes constant time. ■

Next we consider the translation from $\mathcal{L}_A$ to $\mathcal{L}_M$. It is not sufficient to simply replace each Allen predicate with its definition according to the meaning postulates, because the resulting formula is not necessarily in $\mathcal{L}_M$. Indeed, we can show that the problem is inherently intractable:

Theorem 2 *Computing the strongest set of simple metric constraints equivalent to a set of simple Allen constraints is NP-Hard.*

Proof: Checking the consistency of a set of formulas in $\mathcal{L}_A$ is NP-Complete, but checking consistency of formulas in $\mathcal{L}_M$ is polynomial. Since the best translation must preserve consistency, the translation itself must be NP-Hard. ■

Suppose, however, we wish to compute the minimal network representation of a set of simple Allen constraints for other reasons. We can then quickly compute the strongest set of simple metric constraints implied by that network, by computing the metric constraints one Allen constraint at a time. Figure 4 presents the algorithm **Allen-to-metric** that performs this calculation; the following theorem states that this algorithm is optimal.

Theorem 3 *The algorithm* **Allen-to-metric** *is correct and entails minimal loss of information: For any $A \in L_A$, $M \in L_M$, it's the case that $A \models_{\mathcal{L}} M$ iff* **Allen-to-metric**$(A) \models_{\mathcal{L}} M$. *The algorithm runs in $O(e + n^2)$ time, where e is the time needed to compute the minimal network representation of the input, and n is the number of intervals.*

function Allen-to-metric(A) =
input: a simple Allen constraint network A
output: the strongest set of simple metric constraints
implied by A.

let A' be the minimal network representation of A
if A' is inconsistent **then return** any inconsistent metric network
$M_A := \emptyset$
for each pair of intervals i, j **do**
let R be the (complex) Allen relation such that $i(R)j$ appears in A'
$S := \{\ m \mid m$ is of the form $x - y < 0$ or $x - y \leq 0$
and $x, y \in \{i_L, i_R, j_L, j_R\}\ \}$
for each primitive Allen relation r in R **do**
$S := S \cap \{\ m \mid m$ is a simple metric constraint
implied by $i(r)j\ \}$
end do
$M_A := M_A \cup S$
end do
return M_A
end Allen-to-metric

Figure 3: Converting simple Allen constraints to simple metric constraints.

Proof: At the end of the inner loop, it is clear that $m \in S$ iff m is a metric constraint implied by each $i(r)j$ for each $r \in R$; that is, $m \in S$ iff m is implied by $i(R)j$. Since $A \models_{\mathcal{L}} A' \models_{\mathcal{L}} i(R)j$ for each such $i(R)j$ that appears in M, it follows that $A \models_{\mathcal{L}}$ **Allen-to-metric**(A). Therefore the algorithm is correct: if **Allen-to-metric**(A)$\models_{\mathcal{L}} M$, then $A \models_{\mathcal{L}} M$.

To show that the algorithm entails minimal loss of information, suppose that $A \models_{\mathcal{L}} M$. Because M is conjunction of simple metric constraints, without loss of generality we can assume it is a single such constraint: $x - y < n$ or $x - y \leq n$, where $x, y \in \{i_L, i_R, j_L, j_R\}$. Furthermore, because A is equivalent (using the meaning postulates) to a boolean combination of difference inequalities containing only the number 0, it is plain that n cannot be negative; and furthermore, if n is positive, A must also imply the constraint $x - y \leq 0$. So without loss of generality we can also assume that M is of the form $x - y < 0$ or $x - y \leq 0$.

At the start of the loop in which the algorithm selects the pair of intervals $\langle i, j \rangle$ the variable S contains M, and we claim that S must still contain M at the conclusion of the inner loop. Suppose not; then there is some $r \in R$ such that $i(r)j$ has a model $\mathcal{M}$ in which M does not hold. But because $i(R)j \in A'$ and A' is the minimal network representation of A, it must be the case the A has a model that agrees with $\mathcal{M}$ on the interpretations of i and j. Therefore A has a model that falsifies M, so A cannot imply M after all.

The complexity $O(e + n^2)$ follows immediately from the iteration of the outer loop; everything inside takes constant time. ■

Finally we turn to an analysis of the algorithm **combined-metric-Allen**. What is its computational complexity? The answer depends on how many times the algorithm iterates between the two networks. Because each iteration must strengthen at least one simple Allen constraint (which can only be done 13 times per constraint), in the worst case the number is linear in the maximum size of the Allen network (or $O(n^2)$ in the number of intervals). In fact, this is its lower bound, as well: we have discovered a class of temporal reasoning problems that shows that the maximum number of iterations does indeed grow with the size of the constraint set [Kautz and Ladkin, 1991].

Theorem 4 *The algorithm* **combined-metric-Allen** *is correct:* $M \cup A \models_{\mathcal{L}}$ **combined-metric-Allen**(M, A). *The algorithm terminates in* $O(n^2(e + n^3))$ *time, where* n *is the number of intervals that appear in* $M \cup A$, *and* e *is the time required to compute the minimum network representation of* A.

The question of whether **combined-metric-Allen** is a complete inference procedure for the language $\mathcal{L}_M \cup \mathcal{L}_A$ remains open. We are currently investigating whether the algorithm detects all inconsistent networks, and whether it always computes the minimal network representation in $\mathcal{L}_M \cup \mathcal{L}_A$.

5 Conclusions

The framework presented in this paper unifies the great body of research in AI on metric and qualitative temporal reasoning. We demonstrated that both STCSP's and Allen's temporal calculus can be viewed as sublanguages of a simple yet powerful temporal logic. We provided algorithms that translate between the languages with a minimal loss of information. Along the way we generalized known techniques for dealing with non-strict linear inequalities to handle strict inequalities as well. Finally, we showed how the translations can be used to combine two well-understood constraint-satisfaction procedures into one for the union of the two languages.

References

[Aho *et al.*, 1976] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Co., Reading, MA, 1976.

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

[Allen, 1991] James Allen. Planning as temporal reasoning. In *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning (KR-89)*, Cambridge, MA, 1991.

[Dean and McDermott, 87] T. L. Dean and D. V. McDermott. Temporal data base management. *Artificial Intelligence*, 32:1–55, 87.

[Dechter *et al.*, 1989] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning (KR '89)*, page 83, San Mateo, CA, May 1989. Morgan Kaufmann Publishers, Inc.

[Kautz and Ladkin, 1991] Henry Kautz and Peter Ladkin. Temporal constraint propagation: an exercise in hybrid reasoning. In preparation, 1991.

[Ladkin and Madux, 1987] Peter Ladkin and Roger Madux. The algebra of convex time intervals, 1987.

[Malik and T.O., 1983] J. Malik and Binford T.O. Reasoning in time and space. In *Proceedings of 8th IJCAI 1983*, pages 343–345. IJCAI, 1983.

[Montanari, 74] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 74.

[Valdes-Perez, 1986] Raul E. Valdes-Perez. Spatio-temporal reasoning and linear inequalities. A.I. Memo No. 875, M.I.T. Artificial Intelligence Laboratory, Cambridge, MA, February 1986.

[van Beek and Cohen, 1989] Peter van Beek and Robin Cohen. Approximation algorithms for temporal reasoning. Research Report CS-89-12, University of Waterloo, Waterloo, Ontario, Canada, 1989.

[Vilain *et al.*, 1989] Marc Vilain, Henry Kautz, and Peter van Beek. Constraint propagation algorithms for temporal reasoning: a revised report. In Johan deKleer and Dan Weld, editors, *Readings in Qualitative Reasoning About Physical Systems*. Morgan Kaufmann, Los Altos, CA, 1989.

Model-Based Temporal Constraint Satisfaction[†]

(Extended Abstract)

Armen Gabrielian
Thomson-CSF, Inc.
630 Hansen Way, Suite 250
Palo Alto, CA 94304
(415)494-8818
gabrielian%tcipro.uucp@unix.sri.com

Problems in temporal constraint satisfaction arise in a system with interacting elements, where temporal constraints exist (1) between distinct events, (2) on the truth or falsehood of states or partial states, and (3) on responses to external inputs in terms of deadlines. The availability of a causal model or a model-based representation for such a system can provide significant aid in understanding its dynamic behavior and in formulating the mathematical forms of associated temporal constraint satisfaction problems. This is particularly so when the causal model can also serve as a system specification in which logical and temporal properties of systems can be verified formally. A "hierarchical multi-state (HMS) machine" consists of a high-level automaton that is integrated with a temporal interval logic called TIL to provide a unified framework for specification, verification and reasoning for real-time systems. In this paper, an overview of the methodology for addressing very general temporal constraint satisfaction problems arising from the use of HMS machines as causal models of dynamic systems is presented. Specifically, a symbolic execution method is presented for creating schedules for sequential or partially-ordered plans to satisfy complex sets of temporal constraints. Given the HMS machine model of a system and a plan, the method derives the set of mathematical inequalities that potential schedules for the plan must satisfy.

1. Introduction

An executable formal specification of a real-time system can serve as a causal model in which various problems relating to verification of logical and temporal properties can be investigated. Such formal specifications are usually deterministic structures in which responses to all actions are defined precisely. Nondeterministic actions in specifications merely denote alternative actions, all of which are acceptable. Temporal constraint satisfaction, on the other hand, arises in situations where a system is *underspecified*, while its global behavior is constrained in terms of a set of temporal constraints. Thus, in order to apply a causal model to problems of temporal constraint satisfaction, a truly nondeterministic model of behavior is required, in which not all behaviors are necessarily acceptable.

The concept of "hierarchical multi-state (HMS) machine" provides a unified framework for specification, verification and reasoning for real-time systems based on the integration of high-level automata and a temporal logic [5], [4], [6], [3], [2], [7] and [8]. The underlying automaton model of HMS machines reduces the state space by orders of magnitude compared to traditional automata, with explicit modeling of concurrency at different granularities of time. This offers a rich formalism for defining conditional behavior and causal interactions among events and states in both the forward and backward direction. In the forward direction, the past and the present determine the current set of actions. In the backward direction, the future determines the course of present actions. The latter situation can arise in modeling intentionality or in incomplete causal models in which details are omitted.

[†] This work was supported in part by the Office of Naval Research under Contract N00014-89-C-0022.

In HMS machines, nondeterminism is used as a key method for modeling abstraction and generalization. A nondeterministic HMS machine defines a "class" of behaviors. These are instantiated and constrained in terms of higher-level "policy" HMS machines that define dynamic goals and heuristic guidelines for achieving them. Within this framework, planning is accomplished by searching through nondeterministic transitions to determine paths that satisfy both high-level and low-level constraints. A "schedule" for a plan is then defined in terms of delays between adjacent sets of parallel actions. Our purpose here is to indicate how, given such a representation scheme, very general temporal constraint satisfaction problems can be solved by the analysis of HMS-based causal models of real-time systems.

In relating to previous work on temporal constraint satisfaction, we note two differences with [1]. First, in [1] it is assumed that the mathematical formulation of temporal constraints is given and the emphasis is on the development of efficient solution techniques. In contrast, we address the complementary issue of *deriving* the necessary mathematical constraints from the underlying model of the system under consideration. Secondly, only sets of constraints of the form

$$(a_1 \leq X_j - X_i \leq b_1) \vee \ldots \vee (a_n \leq X_j - X_i \leq b_n)$$

are considered in [1]. In our formulation, much more general types of constraints can be derived.

In Section 2 we provide a very brief overview of HMS machines and in Section 3 we present the outline of our scheduling method for temporal constraint satisfaction. Details can be found in the references.

2. Overview of HMS Machines

Both in real-time systems theory and in artificial intelligence studies, various formalisms for specification and modeling of real-time systems have been proposed. However, since the goal of systems theory and AI are rather different, very little effort has been made in integrating such efforts. Hierarchical multi-state (HMS) machines provide a rich framework for specification, verification and reasoning for real-time systems that can be a vehicle for bridging this gap. In the simplest version, an HMS machine is an automaton in which (1) a state can be hierarchically or recursively expanded into an HMS machine itself, (2) multiple states can be active, (3) multiple transitions can fire simultaneously, and (3) transitions are controlled by predicates in a propositional temporal interval logic called TIL. This provides a formal and visual formalism for representing causal interactions among the elements of a complex real-time system in a natural manner. In addition, it overcomes the inability of pure propositional temporal logic in expressing some simple regular properties such as "state A will be true every 5 moments in the future."

HMS machines can be used to define the dynamic behavior of complex real-time systems at various levels of abstraction. Such a representation can be verified for correctness using either correctness-preserving transformation techniques [2] or model checking [7], [4]. Using probabilistic extensions, planning and situation understanding can also be studied under a wide set of uncertainty conditions. Other extensions of HMS machines include algebraic machines in which multiple entities (agents, resources, etc.) can be modeled explicitly, continuous time machines and asynchronous composition of machines with different clock rates.

3. Temporal Constraint Satisfaction of HMS Plans

Problems of temporal constraint satisfaction arise in a nondeterministic HMS machine in attempting to search for a "plan" that satisfies both a global goal and all the local logical and temporal constraints defined on transitions in the language TIL. A sequential plan $p = g_1 g_2 ... g_n$ in our formalism consists of a sequence of sets of transitions, where each set g_i of transitions is assumed to be fired simultaneously. An example of a partially ordered plan is $p = (g_1 ((g_2 g_3) \,||\, (g_4 g_5 g_6)))$, where g_1 can be considered as the root of a tree with two branches, one containing the sequential plan $g_2 g_3$ and the other containing the sequential plan $g_4 g_5 g_6$.

In our approach, we assume a separation between planning and scheduling, with planning being concerned with deriving a plan of non–empty sets of transitions that, ignoring local constraints, can lead a machine from an initial set of states to a desired set of states. Scheduling, on the other hand, is concerned with defining delays between adjacent steps of a plan that allow local constraints to be satisfied. We illustrate our method by considering a simple sequential plan for the HMS machine of Figure 3.1. In this figure, rectangles represent states, dark arrows represent transitions (with * indicating nondeterminism) and thin arrows from states to transitions define "controls" on transitions, with temporal constraints indicated next to encircled T's. Thus, there are no constraints on the transition y, while there are three constraints on the transition x: (1) state A must have been true in the interval [–2, 0], C must have been true in [–3, –2], and (3) D must have been true in [–1, 0]. Here we assume a discrete model of time with 0 indicating the current moment and for each interval the end–points are included. Assuming that the state A and C were true originally, we wish to find a plan consisting of a sequence of transitions and delays that will cause the state B to be true. Note that in TIL $[t_1, t_2]$ and $<t_1, t_2>$ are interval–based generalizations of the temporal logic operators □ ("always") and ◇ ("sometime").

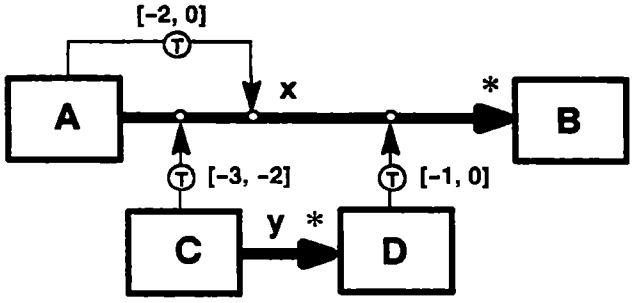


Figure 3.1. A Simple HMS Machine Example for Temporal Constraint Satisfaction

Formally, we represent by $p' = y\,x$ the "potential plan" for achieving our goal. This could be derived by a search process or using heuristic knowledge. To determine the schedule for p′ to satisfy the local constraints, we define the following "variable delay plan"

$$p = \phi^i\, y\, \phi^j\, x,$$

where ϕ stands for a no action or "wait," so that the plan p consists of waiting i moments, firing the nondeterministic transition y, waiting j moments and finally firing the nondeterministic transition x. Our goal is to find a solution to the parametric delays in p that satisfies the three temporal constraints on x. Our solution method is based on a symbolic execution of the plan p to determine parametric facts that must be true during the execution of p. Thus, e.g., after waiting i moments and firing y, we can

assert $[-i-1, 0]A \wedge [-i-1, -1]C \wedge D \wedge \neg C$. Continuing in this manner, we derive the parametric facts that will be true just before executing the transition x. By comparing with the controls that x *must* satisfy, we derive the following set of inequalities:

$$-i-j-1 \leq -2, \quad -i-j-1 \leq -3, \quad -j-1 \leq -2, \quad -j \leq -2.$$

By solving these inequalities, we obtain the generic solution

$$p = \phi^{i>0}\, y\, \phi\, x.$$

Thus, to accomplish our goal, we can wait one or more moments, execute y, wait exactly one more moment and then execute x. Such a sequence will satisfy all the local temporal constraints and achieve the global objective of reaching state B. A larger example with details of the method appears in [6].

Our temporal constraint satisfaction has been generalized both to partially ordered plans and to the continuous time case. Thus, a very general model-based framework is provided for defining causal models of complex real-time systems through which temporal constraint satisfaction problems can be investigated in a systematic manner.

Acknowledgments

The author is indebted to M.K. Franklin for his contributions in the development of the original constraint satisfaction method for sequential plans.

References

[1] Dechter, R., I. Meira, and J. Pearl, "Temporal constraint networks," *Proc. First Int. Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Canada, May 1989.

[2] Franklin, M.K., and A. Gabrielian, "A transformational method for verifying safety properties in real-time systems," *Proc. 10th Real-Time Systems Symposium*, Santa Monica, CA, December 5-7, 1989, pp. 112-123.

[3] Gabrielian, A., "Reasoning about real-time systems with temporal interval logic constraints on multi-state automata," *Proc. Space Operations, Applications and Research Symposium*, Albuquerque, NM, June 26-28, 1990, to appear.

[4] Gabrielian, A., "HMS machines: a unified framework for specification, verification, and reasoning for real-time systems," in *Foundations of Real-Time Computing: Formal Specifications and Methods,* Andre van Tilborg, Ed., Kluwer Acad. Publishers, Norwell, Mass., 1991.

[5] Gabrielian, A., and M. K. Franklin, "State-based specification of complex real-time systems," *IEEE Real-Time Systems Symposium*, 1988, pp. 2-11.

[6] Gabrielian, A., and M.K. Franklin, "Multi-level specification and verification of real-time software," *12th Int. Conf. on Software Engineering*, March 26-30, 1990, Nice, France, pp. 52-62. Revised version to appear in *Communications of the ACM*, March 1991.

[7] Gabrielian, A., and R. Iyer, "Integrating automata and temporal logic: a framework for specification of real-time systems and software," *Proc. Unified Computation Laboratory,* Institute of Mathematics and its Applications, Stirling, Scotland, July 3-6, 1990, to appear.

[8] Gabrielian, A., and M.E. Stickney, "Hierarchical representation of causal knowledge," *Proc. WESTEX-87 IEEE Expert Systems Conference*, June 1987, pp. 82-89.

TEMPORAL REASONING WITH MATRIX-VALUED CONSTRAINTS

Robert A. Morris and Lina Al-Khatib
Dept. of Computer Science
Florida Institute of Technology
150 W. University Blvd.
Melbourne, FL. 32901
morris@cs.fit.edu

1. INTRODUCTION

This paper describes research leading to a modification of the constraint propagation algorithm for applications involving qualitative reasoning about temporal intervals, as defined by James Allen [1]. The work is part of a plan to formally represent the idea of events whose occurrences contain gaps in time. The basic representation mechanism involves countenancing the notion of a collection of (union of [4]) convex (gapless) intervals. The need for an efficient representation of binary temporal relations between these structured objects has led to introducing matrices of binary convex temporal relations. The values of these matrices depict relations among the subintervals of the collections of intervals. The result is a system for solving constraint satisfaction problems involving unions of convex intervals and matrix-valued binary relations between them. An evaluation of the complexity of a constraint propagation algorithm for a network of matrix-valued binary relations shows an expected increase in runtime; however, the benefits of the matrix approach include a more structured knowledge base, which may improve the performance of the constraint problem solving mechanism.

2. BACKGROUND

James Allen's *interval*-based representation of time has been an influential approach to representing qualitative temporal information. The atoms of this calculus is a set of thirteen primitive binary temporal relations on convex intervals, consisting of during (d), precedes (p), starts (s), finishes (f), overlaps (o), meets (m), equals (=), and their converses, preceded by ($\tilde{p}$), etc. (The "converse" of a relation R between i and j is the corresponding temporal relation between j and i; thus "after" is the converse of "before", etc.) A "composition table" (Allen, [1], not reprinted here), allows for the deduction of relations between arbitrary pairs of intervals x and z, based on known relations between x and another interval y, and between y and z. This inferencing is commonly known as "composing" two relations.

Allen's system is amenable to constraint satisfaction techniques for finding solutions to planning and scheduling problems (Mackworth,[7], Mackworth and Freuder, [8], Dechter and Pearl, [2]). In temporal reasoning applications, the nodes of interval constraint networks contain temporal interval values, and the arcs consist of *vectors* (sets) of binary temporal relations between the nodes. The temporal constraint satisfaction algorithm used by Allen [1] (Figure 1) operates on a network of convex intervals and their relations by propagating the effects of updating the temporal knowledge (binary relations) between two convex intervals to the entire network. Knowledge is updated by reducing the set of possible temporal relations between two intervals. Updating is performed by indexing the transitivity table to find the set S of constraints between X and Z, given the X-Y and Y-Z constraints, and then intersecting S with the old value for the set of constraints between X and Z (if one exists). If the intersection contains one element, then the temporal relation between X and Z has been uniquely determined; if the intersection is empty, the network has been shown to be inconsistent.

```
Procedure Propagate(i,j)
begin
  for each interval k in Intervals do
  begin
    temp := Table[i,k] + (Table[i,j] * Table[j,k]);
    if Temp = {} then Signal Contradiction;
    if Table[i,k] <> Temp then Place pair <i,k> on Queue;
    Table[i,k] := Temp;
    Temp := Table[k,j] + (Table[k,i] * Table[i,j]);
    If Temp = {} then Signal Contradiction;
    If Table[k,j] <> Temp then Place pair <k,j> on Queue;
    Table[k,j] := Temp;
   end;
end;
```

Figure 1. Constraint Propagation Algorithm

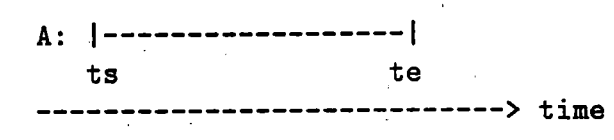
It has been shown (Villain et. al.,[14]) that this propagation algorithm runs in cubic time. As the number of intervals in the network grows, this speed becomes intolerable. Furthermore, as Allen himself noted, the algorithm is incomplete; i.e., it does not always find all the constraints implied by a given network. Complete versions of this algorithm run in exponential time, leading to a proof (Villain et. al, [14]) that the problem of computing network closure is NP-complete.

A number of attempts have been made by Allen and others to improve this situation in various ways (e.g., [3]). For example, restructuring the constraint knowledge base to make it more hierarchical makes its navigation more efficient. The work presented here is in the spirit of these attempts at making networks more navigable. Later, we discuss informally how the use of matrices could lead to efficient techniques to determine constraint network consistency.

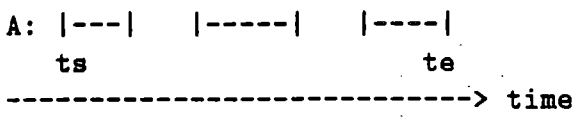
3. UNIONS OF CONVEX INTERVALS

A number of researchers have recently recognized the need for defining a reference structure for non-convex intervals [4], [6]. In particular, Peter Ladkin (Ladkin, [4]) has defined a taxonomy of binary relations between intervals with gaps. Intervals with gaps are useful in two distinct referring contexts. First, these to describe tasks and events which correspond to a recurring time period, such as described by the the referring expressions *Mondays*, *Weekly Faculty Meetings*, etc. Secondly, collections of convex intervals seem to be involved when referring to a single event that is interrupted or suspended in time, and later resumed, e.g., as suggested by the meaning of the sentence "My trip to Orlando was interrupted three times".

In visual terms, such contexts suggest extending the domain of discourse to include not just convex intervals, e.g.:



with endpoints ts and te, but also to allow for bundles of convex intervals containing "gaps":

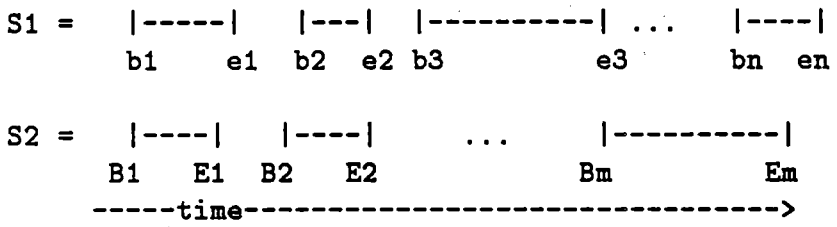


Our claim is that removing the convexity constraint from the characterization of intervals results in a useful way of representing certain types of temporal information that is difficult or impossible to represent assuming convexity.

More formally, in a convex interval-based approach, an interval is represented by an ordered pair $\langle i, j \rangle$ of time units such that $i < j$. What we call, after Ladkin, a *union of convex intervals*, is a special kind of set of convex intervals, viz. a set where the m th j element is less than the m+1 th ith element: $\{\langle i_m, j_m \rangle : 1 \leq m \leq k \& (\forall n)(1 \leq n < k \rightarrow j_n < i_{n+1}\}$. where "$\leq$" and "$<$" denote temporal precedence. Actually, unions of convex intervals are sequences of convex intervals, in that it is assumed that there are functions which extract the first, last, and, in general, the ith subinterval of the collection.

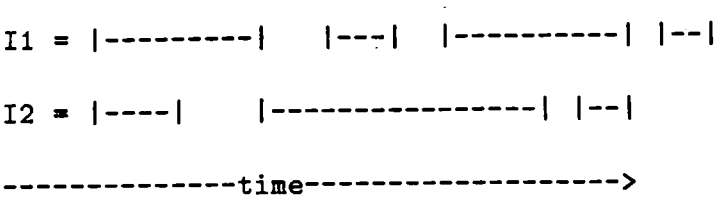
Unions of convex intervals possess an internal structure that is not present in convex intervals. This internal structure allows for a distinction between two kinds of relation that hold between pairs of them, which we designate as *external* and *internal* relations. The following example illustrates each of them.

Let S1 and S2 be two unions of convex intervals defined as follows :



Informally, the external relation between S1 and S2 is the relation between the two convex intervals $< b1, en >$ and $< B1, Em >$. The external relation is important when the interest is strictly on the boundary relations between two unions of convex intervals (e.g., it may be of interest only that S1 is started by S2). By contrast, an internal binary relation is a binary relation between a convex subinterval in S1 and a convex subinterval in S2 or visa versa. Obviously, if S1 has n subintervals and S2 has m, then there are 2nm internal relations between S1 and S2. Elsewhere [11] we have shown how to extract the external binary relations between unions of convex intervals by considering their "boundary subinterval relations", e.g., the relations between subintervals $< b1, e1 >$ of S1 and $< B1, E1 >$ of S2, and similarly between $< bn, en >$ and $< Bm, Em >$.

The internal and external binary relations among intervals i and j with gaps are to be found by a pair-wise examination of all the binary relations among i's and j's subintervals. To bind these pairwise relations together in a common data structure, a matrix notation can be used. The rows of the matrix are the subintervals of i and the columns the subintervals of j. Let $M_{k,l}$ be the internal relation between subinterval k of i and l of j, for each pair of subintervals of i and j. For example, consider the following pair of intervals:



The matrix of internal relations between I1 and I2 is the following:

$$M = \begin{vmatrix} \tilde{s} & m & p \\ \tilde{p} & d & p \\ \tilde{p} & \tilde{o} & \tilde{f} \\ \tilde{p} & \tilde{p} & \tilde{p} \end{vmatrix}$$

Thus, for example, the relation between the third subinterval of I1 in the figure above and the second subinterval of I2 is found by referencing $M_{3,2} = \tilde{o}$.

This representation can be viewed as analogous to the use of matrices of numbers to depict the coefficients of terms in a set of linear equations. In this case, the corresponding expression of which the matrix is a representation is a first-order predicate calculus expression of the form: $P_{1,1}(x_1, y_1), P_{1,2}(x_1, y_2), ..., P_{2,1}(x_2, y_1), ..., P_{n,m}(x_n, y_m)$, where each $P_{i,j}$ represents a set of binary relations among Allen's thirteen atoms, and for each pair x_i, x_{i+1} (y_j, y_{j+1}), $1 \leq i(j) \leq m(n)$, the relation between these pairs is precedes. Each matrix is, thus, a constraint network, but one of a special kind, characteristic of relations among collections of convex intervals which are (possibly gapped) subintervals of larger intervals.

4. CONSTRAINT REASONING WITH UNIONS OF CONVEX INTERVALS

As noted earlier, there are two distinct reasons for offering an extension to Allen's convex interval calculus to include collections of convex intervals. First, it seems to provide a more natural representation of the referring mechanism for temporal intervals, and second it seems to fit into the spirit of recent attempts to improve the cubic time and quadratic space requirements of Allen's original approach. In particular, combining convex intervals together into a union of convex intervals captures much of the content of the concept of a reference interval [1], since a union of convex interval is a collection of semantically related subintervals. Unlike Allen's reference interval idea, our approach allows for an explicit representation, in terms of matrices, of binary relations between collections of subintervals within a constraint network. In this representation, there is a reduction of the space requirements (since the order of the subintervals are represented by the position of the subinterval in the matrix, and not by an arc), without a loss of information.

We have extended Allen's definition of relation composition to accommodate the new matrix representation of binary relations. This extension allows for an application of the constraint propagation techniques described earlier in this paper to a network of gapped intervals and binary relations between them.

The composition method we propose is a generalization of the method of composition proposed by Allen for convex intervals. Given two interval matrices $P^{n,m}$ and $R^{m,k}$ of dimensions $n*m$ and

$m * k$ respectively, represented as follows:

$$P = \begin{vmatrix} P_{1,1} & P_{1,2} & P_{1,3} & \ldots & P_{1,m} \\ P_{2,1} & P_{2,2} & P_{2,3} & \ldots & P_{2,m} \\ \vdots & & & & \\ P_{n,1} & P_{n,2} & P_{n,3} & \ldots & P_{n,m} \end{vmatrix}$$

$$R = \begin{vmatrix} R_{1,1} & R_{1,2} & R_{1,3} & \ldots & R_{1,k} \\ R_{2,1} & R_{2,2} & R_{2,3} & \ldots & R_{2,k} \\ \vdots & & & & \\ R_{m,1} & R_{m,2} & R_{m,3} & \ldots & R_{m,k} \end{vmatrix}$$

the composition operation results in a matrix $P \circ R$ comprising the following:

$$Q = P \circ R = \begin{vmatrix} Q_{1,1} & Q_{1,2} & Q_{1,3} & \ldots & Q_{1,k} \\ Q_{2,1} & Q_{2,2} & Q_{2,3} & \ldots & Q_{2,k} \\ \vdots & & & & \\ Q_{n,1} & Q_{n,2} & Q_{n,3} & \ldots & Q_{n,k} \end{vmatrix}$$

where $Q_{i,j} = \bigcap_{l=1}^{m} P_{i,l} \circ R_{l,j}$. (This composition operation is similar the operation defined in the binary constraint networks of Ladkin (Ladkin and Maddux, [5]) for solving CSPs (also Maddux, [9])). The resulting matrix contains vectors (sets) of atomic binary relations representing constraints between pairs of subintervals of the two unions of convex intervals. As before, composing two matrices represents a single act of qualitative reasoning about intervals; the only difference is that the intervals are allowed to contain gaps.

As mentioned, this definition of composition leads directly to a modification of the constraint propagation technique for reasoning with collections of intervals. Recall that binary constraints are depicted as vectors of relations between convex intervals. Given two matrices, M^1 and M^2, we define $M^1 + M^2$ as follows: let $M^n_{i,j}$ be the vector of binary relations between subintervals i and j of two unions of convex intervals I and J constrained by M^n. Then $M^1 + M^2$ is the matrix which results from performing vector additions $M^1_{i,j} + M^2_{i,j}$, for all i, and j in M^1 and M^2. (Matrix addition is always between 2 matrices of the same dimensions, and the result is a matrix formed by intersecting the corresponding values at each position). Matrix multiplication corresponds to the matrix composition operation defined above.

With these operations defined, we can now apply the constraint propagation algorithm (Figure 1) directly to a network of unions of convex intervals. In this case, Table[i,j] will consist of a matrix of binary relations between unions of convex intervals i and j. For example, consider the following constraint network:

	A	B	C
A	M0	M1	M2
B	M4	M0	M3
C	M5	M6	M0

where M0 is the matrix where all its elements are the equality relation, and the other values are defined as follows:

$$M1 = \left| \begin{array}{ccc} \tilde{p}\tilde{m} & o\tilde{f} & p \\ \tilde{p} & \tilde{p}\tilde{o}\tilde{m} & o \end{array} \right|$$

$$M2 = \left| \begin{array}{cc} \tilde{p}\tilde{m} & opm \\ \tilde{p} & \tilde{p}\tilde{d}\tilde{o}\tilde{m}\tilde{s}dos \end{array} \right|$$

$$M3 = \left| \begin{array}{cc} \tilde{d}\tilde{f} & p \\ \tilde{p} & so \\ \tilde{p} & \tilde{p}\tilde{m}d \end{array} \right|$$

M4 = INV M1
M5 = INV M2
M6 = INV M3

The inverse of a matrix M of dimension $m * n$, INV $M_{m*n} = M'_{n*m}$ where for each element $m'_{i,j}$ in M', $m'_{i,j} = \tilde{m_{j,i}}$, $m_{j,i}$ in M. Again, each of the Mi represents constraints between pairs of unions of convex intervals from a network of three unions of convex intervals, A, B and C. For example, M2 represents the matrix of constraints between unions of convex intervals A and C, each containing two convex subintervals. M2 says that subinterval 1 of A must either be preceded by or met by subinterval 1 of C, and must either overlap, precede, or meet subinterval 2 of C (and so on for each of the other values of M2).

Suppose now that the constraint network is updated by adding a binary constraint between A and C, expressed as the following matrix:

$$R(A,C) = \left| \begin{array}{cc} \tilde{p}\tilde{m}\tilde{o} & pm\tilde{f} \\ \tilde{p} & dso \end{array} \right|$$

The effects of revising the constraint network based upon this addition can be summarized as follows:

	A	B	C
A	M0	M1'	M2'
B	M4'	M0	M3'
C	M5'	M6'	M0

where

$$M1' = \left| \begin{array}{ccc} \tilde{p}\tilde{m} & o & p \\ \tilde{p} & \tilde{p}\tilde{o}\tilde{m} & o \end{array} \right|$$

$$M2' = \begin{vmatrix} \tilde{p}\tilde{m} & pm \\ \tilde{p} & dos \end{vmatrix}$$

$$\text{M3'} = \begin{vmatrix} \tilde{d}\tilde{f} & p \\ \tilde{p} & o \\ \tilde{p} & d \end{vmatrix}$$

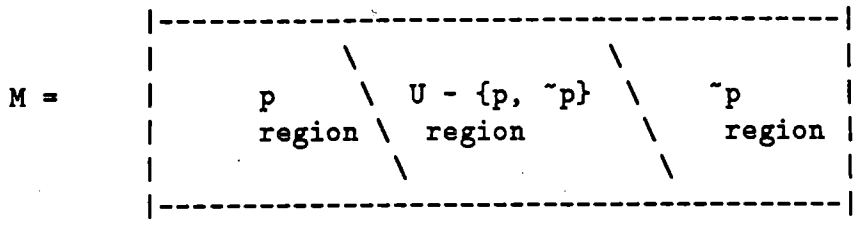
M4' = INV M1'
M5' = INV M2'
M6' = INV M3'

5. PERFORMANCE AND IMPROVEMENTS

Although the propagation algorithm, as modified to perform updates on matrix-valued constraints, becomes more complex, the network on which it operates becomes smaller than it would have been if all the subintervals of a union of convex intervals had been depicted as separate nodes. Instead, qualitative temporal information about gapped reference intervals is compressed within the matrix.

Assuming, for the sake of simplicity, that the constraint matrices are all square $(m * m)$ in size, and that the network contains k unions of convex intervals, it can be shown that the runtime of the algorithm is still in polynomial range, specifically, $O(k^3(m^5))$. The reasoning is as follows [13]. The procedure propagate is called $O(k^2)$ times to compute closure. That is, a total of $O(k^2)$ pairs of unions of convex intervals $\langle i, j \rangle$ can appear on Queue. Each value of the constraint matrix can change at most 13 times, and there are m^2 of these; hence, each pair $\langle i, j \rangle$ can appear at most $O(13m^2)$ times on Queue. For each Queue element, the amount of processing required for the constraint network is $O(k(m^2 + m^3))$. Thus, the total cost to obtain closure for a network of gapped intervals and matrix relations is $O(k^2(13m^2)k(m^2 + m^3))$, which is $O(k^3(m^5))$.

An advantage of the compressed network of unions of convex intervals over the full network of convex intervals is in the ability to efficiently examine the matrices to discern patterns of relations which any consistent matrix must exemplify. Fast algorithms for verifying matrix consistency can be employed as a preprocessing stage in the solving of constraint reasoning problems. For example, let us consider the case of a completely determined matrix (one we shall call an *atomic* matrix), which is one all of whose values consist of a single element from one of Allen's thirteen atoms (this set we call U). We can say that, in general, a consistent matrix will contain rows each one of which will consist of 0 or more instances of the relation p followed by 0 or more occurrences of any relation in the set $U - \{p, \tilde{p}\}$ followed by 0 or more occurrences of $\tilde{p}$. Furthermore, the number of occurrences of p in each row cannot decrease as the row index increases. Correspondingly, the number of occurrence of $\tilde{p}$ cannot increase as the row index increases. Consequently, rules regarding the patterns of a consistent matrix can be defined as being comprised of three regions of binary relations: the left triangular p region, the middle $U - \{p, \tilde{p}\}$ region, and the right $\tilde{p}$ region. More graphically, each matrix must have the following pattern:



Future research will include the development and formulation of a complete collection of rules for consistent matrices of binary temporal relations, which can be used in the construction of improved algorithms for solving constraint problems. In addition, it should be possible to find a way of "reducing" matrices into more space efficient formats based on the canonical patterns for consistent matrices just described.

6. SUMMARY

This paper has introduced an approach to constraint temporal reasoning based on unions of convex intervals and binary constraints between them, implemented as matrices of convex binary relations. Such an extension to James Allen's framework can be viewed as an implementation of his notion of a reference interval. A constraint propagation algorithm for networks of matrices of temporal relations was described. Finally, an informal discussion of ways of improving the runtime performance of propagation was presented, based on rules for identifying consistent matrices of temporal relations.

Bibliography

[1] Allen, J. (1983) Maintaining Knowledge About Temporal Intervals. In Brachman, R., and Levesque, H., (eds.) *Readings in Knowledge Representation*, (San Mateo:Morgan Kaufman), 510- 521.

[2] Dechter, R. and Pearl, J. (1988) Network-Based Heuristics for Constraint-Satisfaction Problems. *Artificial Intelligence* 34:1-38.

[3] Koomen, J., (1989) Localizing Temporal Constraint Propagation. In Brachman, R., Levesque, H., and Reiter, R. eds., *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning, KR'89*, pp. 198-202.

[4] Ladkin, P. (1987) Time Representation: A Taxonomy of Interval Relations. In *Proceedings of AIII-86*, (San Matyeo:Morgan Kaufman), 360- 366.

[5] Ladkin, P., and Maddux, R. (1988) Representation and Reasoning With Convex Time Intervals. Kestrel Institute Technical Report KES.U.88.2.

[6] Ligozat, G., (1990) Weak Representation of Interval Algebras. In *Proceedings of the Eighth National Conference on Artificial Intelligence* (AAAI-90), Vol. 2, pp. 715-720.

[7] Mackworth, A. (1977) Consistency in Networks of Relations. *Artificial Intelligence*, 8:99-118.

[8] Mackworth, A., and Freuder, E. (1985) The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence* 25:65-74.

[9] Maddux, R. (1982) Some Varieties Containing Relation Algebras. *Transactions of the American Mathematical Society*, 272:501-526.

[10] Morris, R., and Al-Khatib, L., A Matrix Representation of Events With Gaps. *Proceedings of the Third Florida Artificial Intelligence Research Symposium (FLAIRS-90)*, Cocoa Beach, FL, 1990.

[11] Morris, R., and Al-Khatib, L., *A Temporal Relational Calculus for Gapped Events*, in Temporal and Real-Time Specifications: Collected Papers of the ICSI Workshop, August 9-10, 1990.

[12] Morris, R., and Al-Khatib, L., *An Interval-Based Temporal Relational Calculus For Events With Gaps*. The Journal of Experimental and Theoretical Artificial Intelligence, forthcoming.

[13] Van Beek, P. (1990) Private Correspondence.

[14] Villain, M., Kautz, H., van Beek, P. (1990) Constraint Propagation Algorithms for Temporal Reasoning: A Revised Report. In Weld, D., and de Kleer, J. (eds), *Readings in Qualitative Reasoning About Physical Systems*, (San Mateo:Morgan Kaufmann), 373-381.

Reason Maintenance and Inference Control for Constraint Propagation over Intervals

Walter Hamscher
Price Waterhouse Technology Centre
68 Willow Road, Menlo Park, CA 94025
hamscher@tc.pw.com

1 Overview

ACP is a fully implemented constraint propagation system that computes numeric intervals for variables [Davis, 1987] along with an ATMS label [de Kleer, 1986a] for each such interval. The system is built within a "focused" ATMS architecture [de Kleer and Williams, 1986, Forbus and de Kleer, 1988, Dressler and Farquhar, 1989] and incorporates the following ideas to improve efficiency:

- Since variable values are intervals, some derived variable values may subsume more specific (superset) interval values. Variable values that are subsumed are marked as inactive via a simple and general extension to ATMS justifications. Other systems that maintain dependencies while inferring interval labels either use non-monotonic reasoning [Simmons, 1986, Williams, 1989] or incorporate the semantics of numeric intervals into the ATMS itself [Dague *et al.*, 1990].
- Solving a constraint for a variable already solved for can cause redundant computation of variable bindings and unnecessary dependencies [Sussman and Steele, 1980, de Kleer, 1986b]. ACP deals with this problem by caching with each variable binding not only its ATMS label, but also the variable bindings that must also be present in any supporting environment.
- The user of the constraint system may know that certain solution paths for deriving variable bindings are uninteresting. A unary "protect" operator is incorporated into the constraint language to allow the user to advise ACP to prune such derivation paths.

2 Motivation

ACP is part of the model-based financial reasoning system CROSBY [Hamscher, 1990]. Financial reasoning has long been recognized as an appropriate domain for constraint-based representation and reasoning approaches [Bouwman, 1983, Reboh and Risch, 1986, Apte and Hong, 1986, Lassez *et al.*, 1987, Dhar *et al.*, 1988, Dhar and Croker, 1988, Peters, 1989]. For the most part CROSBY uses ACP in the traditional way: to determine the consistency of sets of variable bindings, and to compute values for unknown variables. For example, CROSBY might have a constraint such as

$$\text{Days.Sales.in.Inventory} = \frac{30 \times \text{Monthly.Cost.of.Goods.Sold}}{\text{Average.Inventory}}$$

Given the values Average.Inventory $\in (199, 201)$ and Cost.of.Goods.Sold $\in (19, 21)$, ACP would compute Days.Sales.in.Inventory $\in (2.84, 3.02)$. Had the fact that Days.Sales.in.Inventory $\in (3.5, 3.75)$ been previously recorded, a conflict would now be recorded.

For the purposes of this paper, all the reader need know about CROSBY is that it must construct, manipulate, and compare dozens to hundreds of combinations of underlying assumptions about the ranges of variables. This motivates the need for recording the combinations of underlying assumptions on which each variable value depends, which in turn motivates the use of an ATMS architecture to record such information. Although there is extensive literature on the interval propagation aspects of the problem, little of the work addresses the difficulties that arise when dependencies must be recorded for the many intermediate results. The poor performance of the obvious implementation strategy motivates the ideas discussed below.

3 Syntax and Semantics

ACP uses standard notation as reviewed here: $[1, 2)$ denotes $\{x : 1 \leq x < 2\}$, $(-\infty, 0)$ denotes $\{x : x < 0\}$, and $[42, +\infty)$ denotes $\{x : 42 \leq x\}$. The symbols $+\infty$ and $-\infty$ are used only to denote the absence of upper and lower bounds; they cannot themselves be represented as intervals. Intervals may not appear as lower or upper bounds of other intervals, that is, $[0, (10, 20)]$ is ill formed. $(,)$ denotes the empty set.

All standard binary arithmetic operators are supported, with the result of evaluation being the smallest interval that contains all possible results of applying the operator pointwise [Davis, 1987]. For example, $[1, 2) + (1, 2]$ evaluates to $(2, 4)$. $(1, 2)/[0, 1)$ evaluates to $(1; +\infty)$, with the semicolon replacing the comma to denote an interval that includes the undefined result of division by zero.

All binary relations r evaluate to one of the constant symbols T or F, obeying the following rule for intervals I_1 and I_2:

$$I_1 \; r \; I_2 \leftrightarrow \exists x, y : x \; r \; y \wedge x \in I_1 \wedge y \in I_2$$

Corollary special cases include $\forall x : x\ r\ (-\infty, +\infty)$, which evaluates to T, and $\forall x : x\ r\ (,)$ which evaluates to F.

Interval-valued variables can appear in expressions and hence in the result of evaluations, for example, evaluating the expression $([1,2] = [2,4] + c)$ yields $c = [-3,0]$. Appealing to the above rule for binary relations, $c = [-3,0]$ can be understood to mean $c \in [-3,0]$.

ACP has a unary "protect" operator, denoted "!". Hence in the expression $(a = !(b + c))$ with a, b, and c being variables, b and c are said to be protected. The effect of protection is that evaluating any expression, all of whose variables are protected, yields T. For example, evaluating $([1,2] = [2,4] + !c)$ yields T. The benefit of this operator is that the ACP user can advise the system not to ever waste effort trying to solve for certain variables. For example, CROSBY constructs linear regression equations of the form $y = \alpha_0 x_0 + \ldots + \alpha_n x_n + \beta$, with α_i and β denoting constants. In this context it makes no sense to try to use the dependent variable y to solve for any of the n independent x_i variables. Protecting the x_i variables is a simple, local, modular way to prevent ACP from doing so, as will be seen below.

4 Recording Dependencies

Recording dependencies and managing multiple contexts complicates interval propagation considerably, because what appear to be intermediate results must be stored permanently, but should not always be used to trigger further inferences. To illustrate this, consider the following example. Node n_0 expresses the relation $(a = b + c)$, true in the empty ATMS environment $\{\}$:

$$n_0 : \quad (a = b + c) \quad \{\}$$

Nodes n_1 and n_2 bind the variables b and c, respectively, under the assumptions named B and C:

$$\begin{array}{ll} n_1 : & (b = (6,9)) \quad \{B\} \\ n_2 : & (c = (10,11)) \quad \{C\} \end{array}$$

Constraint propagation yields a value for a, creating node n_3, justified by justification j_1:

$$\begin{array}{ll} j_1 : & n_3 \leftarrow n_0, n_1, n_2 \\ n_3 : & (a = (16,20)) \quad \{B, C\} \end{array}$$

(In a naive implementation, the system might at this point try to derive values for b or c using the new value of a; this is an instance of "reflection" and ACP's method for preventing it will be discussed in the next section.)

A query for the value of a in the environment $\{B, C\}$ would now return node n_3. Suppose we get a new value for a under assumption A, denoted by node n_4:

$$n_4 : \quad (a = (17,19)) \quad \{A\}$$

Since this value of a is a subset of the interval for a derived earlier, a new justification is required for n_3, with a resulting change to the label of n_3:

$$\begin{array}{lll} j_2 : & n_3 \leftarrow n_4 & \\ n_3 : & (a = (16,20)) \quad \{B,C\}\{A\} & \text{Label update} \end{array}$$

Note that the less specific interval $(16,20)$ for a will always need to be kept around. A query for the value of a in the environment $\{B, C\}$ would still return node n_3, but a query in environment $\{A\}$ should only return node n_4, even though n_3 is true as well. "Shadowing" justifications are introduced to provide this functionality.

A shadowing justification obeys the normal (horn clause) semantics, that is, the consequent is true in any environment in which all its antecedents are true. This criterion results in updates to the environment labels, with only minimal environments being stored in any node label $L(N)$ [de Kleer, 1986a]. However, all nodes also have a "shadow label." Any node supported by a shadowing justification in environment E also has E added to its shadow label $S(N)$, obeying the usual minimality convention. ACP distinguishes between nodes being *true* in an environment, and *active* in an environment with respect to queries and with respect to making inferences:

$$N \text{ is } \textit{true} \text{ in } E \quad \leftrightarrow \quad \exists E_n \in L(N) : E_n \subseteq E$$

$$N \text{ is } \textit{active} \text{ in } E \quad \leftrightarrow \quad \exists E_n \in L(N) : E_n \subseteq E \wedge \neg\exists E_s \in S(N) : E_s \subseteq E$$

Intuitively, shadowing environments make the node invisible in all their superset environments. A node shadowed in the empty environment $\{\}$ would be true in all environments, but no inferences would be made from it.

In the example above, j_2 would be a shadowing justification, since in any environment in which n_4 is true, n_3 should be ignored. Shadowing justifications will be denoted by $\Leftarrow$ and the shadow label as that label appearing to the right of a "\" character. Note that any environment appearing in the shadow label must also be a superset of some environment in the normal label. However, for compactness of notation in this paper, environments that appear in both the normal and shadow label will only be shown to the right of the "\" character. In the example below, the reader should understand that the normal label of n_3 is actually $\{B, C\}$ $\{A\}$:

$$\begin{array}{lll} j_2 : & n_3 \Leftarrow n_4 & \\ n_3 : & (a = (16,20)) \quad \{B,C\} \setminus \{A\} & \text{Label update} \end{array}$$

Since any number of different interval values for a variable can be created in any order, it is in principle possible for $O(n^2)$ shadowing justifications to be installed for a variable with n bindings. However, no node ever need be supported by more than one shadowing justification, so some of these shadowing justifications could be deleted. For example, suppose three nodes n_{101}, n_{102}, and n_{103} are created. The sequence of new justifications and environment propagations illustrates that after j_{102} and j_{103} are created, j_{101} can be deleted:

$$\begin{array}{lll} n_{101} : & x = [0,10] \quad \{X1\} & \\ n_{102} : & x = [4,6] \quad \{X2\} & \\ j_{101} : & n_{101} \Leftarrow n_{102} & \\ n_{101} : & x = [0,10] \quad \{X1\} \setminus \{X2\} & \text{Label update} \\ n_{103} : & x = [2,8] \quad \{X3\} & \text{New node} \\ j_{102} : & n_{103} \Leftarrow n_{102} & \\ n_{103} : & x = [4,6] \quad \{X3\} \setminus \{X2\} & \text{Label update} \\ j_{103} : & n_{101} \Leftarrow n_{103} & \\ n_{101} : & x = [0,10] \quad \{X1\} \setminus \{X2\}\{X3\} & \text{Label update} \end{array}$$

ACP attempts to minimize the number of justifications created by deleting each shadowing justification that is no longer needed. Although deleting justifications is not a normal operation for an ATMS since it can lead to incorrect labelings, this special case guarantees that all environment labels remain the same as if the deletion had not taken place. Since the justifications were redundant, an efficiency advantage accrues from not recomputing the labels as more environments are added.

Having defined the distinction between nodes being *true* versus being *active*, we now turn to methods for controlling propagation inferences.

5 Propagation

ACP propagates interval values for variables using "consumers" [de Kleer, 1986b]. A consumer is essentially a closure stored with a set of nodes; it runs exactly once with those nodes as its arguments the first time they all become true. Normally, a consumer creates a new node and justification whose antecedents are the nodes whose activation triggered it. ACP is built using a focused ATMS that maintains a single consistent focus environment and only activates consumers to run on nodes that are true in that focus environment. ACP takes this a focusing notion step further, running consumers only on nodes that are active.

The propagation mechanism of ACP distinguishes between constraints and bindings. A binding is type of constraint in which only one variable and one interval or constant appears. For example, $n_0 : (a = b + c)$ is a constraint and $n_1 : (b = (6,9))$ and $n_{200} : (a = 2)$ are bindings. Propagation of a constraint node works as shown in the procedure below. For simplicity, the example below shows variables bound only to integers, rather than to intervals as would be done in ACP:

1. When a constraint node becomes active, install consumers to trigger propagation on each of the variables that appear in the constraint. For example, when $n_0 : (a = b + c)$ becomes active, consumers will be installed for variables a, b, and c.
2. When a binding node for a variable becomes active, run each of its consumers; each consumer will substitute the current binding into the constraint and evaluate it. For example, when $n_1 : b = (6,9)$ becomes active, the constraint $n_0 : (a = b + c)$ will be evaluated given n_1, to produce a new constraint $a = (6,9) + c$.
3. The result of the evaluation in step 2 will fall into one of four cases:
   (a) The constant F. For example, if $(a * b = 7)$ and $a = 0$, evaluation returns F. Create a justification for the distinguished node $\perp$ from the current antecedent nodes, which will result in an ATMS conflict.
   (b) The constant T. For example, if $(a * b = 0)$ and $a = 0$ then the evaluation will return T. Do nothing.
   (For another example, if $a = 2$ and $a = !(b + c)$ the evaluation returns T, because all the variables in $2 = !(b + c)$ are protected.)
   (c) A binding. For example, if $a = 2$ and $a = b + 2$ then evaluation returns the binding $b = 0$. Create a new node containing the binding and justify it with the current antecedents.
   (d) A constraint. For example, if $a = 2$ and $a = b + c$ then evaluation returns $2 = b + c$. Go back to step 1 above for the new constraint.

5.1 Solution Trees

The propagation procedure above is straightforward but would in general result in unnecessary work. For one thing, the system is trying to bind all the variables in $a = b + c$ simultaneously, and given $b = 2$ and $c = 2$, would derive $a = 4$ in two different ways. The variables should only be bound in some global strict order (alphabetic, for example) to prevent this. On the other hand, any subexpression that contains operators with idempotent elements does not always require all its variables to be bound before being reduce to a binding; for example, the constraint $a = b * c$, evaluated with $c = 0$, should immediately yield $a = 0$, instead of waiting for a value for b. Finally, since some variables are protected, we can guarantee that certain sequences of bindings and evaluations will never yield any new bindings. Although relatively minor from a purely constraint processing point of view, these are all genuine concerns in ACP because the computational overhead of creating new nodes, justifications, and consumers far outweighs the work involved in actually evaluating the constraints and performing the associated arithmetic operations.

Whenever a new constraint node is created, ACP performs a static analysis to find all the legal sequences in which its variables could be bound. The result of this analysis is represented as a directed tree whose edges each correspond to a variable in the constraint. This is called the solution tree. Each path starting from the root represents a legal sequence. The recursive algorithm for generating this tree simply adds a new arc for each variable appearing in the current expression, from the current root to a new tree formed from the expression derived by deleting that variable. For example, the root of the tree for $(a = b + c)$ has three branches: one for a leading to a subtree that is the tree for $(b + c)$; one for b leading to a subtree that is the tree for $(a = c)$; one for c leading to a subtree for $(a = b)$. In this example the c branch can be pruned, because c (alphabetically) precedes neither a nor b. Had the expression been $(a = b * c)$, the branch could not be pruned, because c could be bound to 0 to produce the binding $a = 0$. On the other hand, had the expression been $(!a = b + c)$, the b branch could have been pruned because the tree for the subexpression $(!a = c)$ consists only of a single branch a, which does not precede b.

The propagator computes the solution tree once and caches it, then step 1 of the propagation procedure presented in the previous section need only install consumers on variables corresponding to branches emanating from the corresponding position in the tree. This additional complexity is worthwhile; it is not unusual in

CROSBY for variables to acquire many different bindings, and it would be wasteful for ACP to repeatedly rediscover worthless sequences of variable bindings.

In an example shown earlier, recall that we had the nodes:

$$\begin{array}{lll} n_0: & (a = b + c) & \{\} \\ n_1: & (b = (6,9)) & \{B\} \\ n_2: & (c = (10,11)) & \{C\} \\ n_3: & (a = (16,20)) & \{B,C\} \end{array}$$

The solution tree ensures that the n_0 and n_1 would have been combined to get $(a = (6,9) + c)$, which would then have been combined with n_2 to get n_3, without deriving the result in the symmetric (and redundant) fashion.

5.2 Reflection

As mentioned earlier, nodes n_0 and n_3 might in principle be combined and evaluated to yield $((16,20) = b + c)$, "reflecting" back through the n_0 constraint to derive new values of b and c. In general, there is little point in attempting to derive values for a variable x using constraints or bindings that themselves depend on some binding of x (the exception occurs when the constraints can be solved by iterative relaxation; ACP does not attempt to solve such cases).

The straightforward and complete method for checking this is to search each of the directed acyclic graphs (DAGs) of justifications supporting any binding about to be combined with a given constraint and propagated. If that constraint appears in every DAG, inhibit propagation. Although it sounds expensive, empirical tests with ACP show that this method is actually worth its cost: compromise strategies – such as searching the DAGs only to a limited depth – often fail to detect reflections because nodes can be supported via arbitrarily long chains of shadowing justifications, and when reflections go undetected, many extra nodes and justifications get created. Depth first traversals of justification trees are fast relative to the costs associated with creating unnecessary bindings.

This strategy can be improved by caching with each node its *essential support set*, and testing that before searching the DAG. The essential support set of a node is that set of nodes that must appear in a justification DAG for any set of supporting assumptions. For example, n_0, n_1 and n_2 all have empty essential support sets; node n_3 has the essential support set $\{n_0, n_1, n_2\}$. ACP tests essential support sets to see whether they contain a binding node for the variable about to be propagated; if so the propagation is inhibited; if not the full DAG search is performed. In the example above, node n_3 does not combine with $n_0 : (a = b+c)$ because n_0 is in its essential support set. Essential support sets can be easily computed locally and incrementally each time a justification is installed, and they have the useful property that once created, they can subsequently only get smaller as propagation proceeds. For example, if some new justification $n_3 \leftarrow n_{201}$ were added, nodes n_0, n_1, and n_2 could be deleted from the essential support set of n_3. In that case n_3 would then appropriately continue to propagate with constraint n_0.

Essential support sets essentially cache the result of searching the support DAG (the basic strategy) ignoring ATMS labels. As compared to the complete and correct strategy, which is to search the DAG, the essential support set strategy can err only by not detecting reflections. In practice the caching of essential support sets is cheap and fast enough to represent a worthwhile preliminary test.

With this additional propagation machinery in place we can now follow ACP as it continues to propagate in focus environment $\{A, B, C\}$ from n_4 as shown below.

$$\begin{array}{lllll} n_4: & (a = (17,19)) & & & \text{New node} \\ j_3: & n_1 \leftarrow n_0, n_2, n_4 & & & \\ n_1: & (b = (6,9)) & \{B\}\{A,C\} & & \text{Label update} \\ j_4: & n_5 \leftarrow n_0, n_1, n_4 & & & \\ n_5: & (c = (8,13)) & \{A,B\} & & \text{New node} \\ j_5: & n_5 \Leftarrow n_2 & & & \\ n_5: & (c = (8,13)) & \{A,B\} \setminus \{C\} & & \text{Label update} \end{array}$$

ACP creates the new node $n_5 : (c = (8,13))$, active only in $\{A, B\}$. Hence, querying ACP for the value of c yields the following results in each of the following environments:

$$\begin{array}{rll} \{\}, \{A\}, \{B\}: & (-\infty, +\infty) & \\ \{A,B\}: & (8,13) & n_5 \\ \{C\}, \{A,C\}, \{B,C\}, \{A,B,C\}: & (10,11) & n_2 \end{array}$$

5.3 Overlapping Intervals

Finally, ACP needs to deal with cases in which a variable is assigned intervals which have nonempty intersections but do not subsets of one another; these are called overlapping intervals. ACP uses shadowing justifications to control the number of overlapping intervals created. Suppose that nodes n_{201} and n_{202} are created with overlapping values $(1,10)$ and $(5,20)$. A third node n_{203} is created to represent their intersection $(5,10)$, and this node in turn shadows the two nodes that support it.

$$\begin{array}{lllll} n_{201}: & (x = (1,10)) & \{X1\} & & \\ n_{202}: & (x = (5,20)) & \{X2\} & & \\ j_{200}: & n_{203} \leftarrow n_{201}, n_{202} & & & \\ n_{203}: & (x = (5,10)) & \{X1,X2\} & & \text{New node} \\ j_{201}: & n_{201} \Leftarrow n_{203} & & & \\ n_{201}: & (x = (1,10)) & \{X1\} \setminus \{X1,X2\} & & \text{Label update} \\ j_{202}: & n_{202} \Leftarrow n_{203} & & & \\ n_{202}: & (x = (5,20)) & \{X2\} \setminus \{X1,X2\} & & \text{Label update} \end{array}$$

Querying ACP for the value of x yields a different result in each of the following environments:

$$\begin{array}{rll} \{\}: & (-\infty, +\infty) & \\ \{X1\}: & (1;10) & n_{201} \\ \{X2\}: & (5,20) & n_{202} \\ \{X1,X2\}: & (5,10) & n_{203} \end{array}$$

Although in the worst case n interval assignments to a variable could result in $O(n^2)$ overlaps, in practice the number of intervals actually created is acceptable. Intuitively speaking, the reason for this is that the propagation strategy ensures that overlapping intervals are only derived from the most restrictive intervals already present in the current focus environment. Furthermore, whenever a new overlapping interval is created, the two intervals that it was created from become shadowed and no more inferences will be drawn from them in the current focus environment.

6 Conclusion

ACP integrates constraint propagation over intervals with assumption-based truth maintenance, contributing the following novel inference control techniques:

- Variable values that are subsumed are marked as inactive via an extension to ATMS justifications, without any need for recourse to non-monotonic reasoning.
- ACP augments its reflection test by caching with each variable binding not only its ATMS label, but also the variable bindings that must be present in any supporting environment.
- A new operator is incorporated into the constraint language to allow the user to advise ACP to prune useless derivation paths.

ACP is implemented in 5500 lines of "Future Common Lisp" on a Symbolics lisp machine. Roughly one half is devoted to the expression evaluator, one third is the focused ATMS, and ACP-specific code comprises the remainder.

References

[Apte and Hong, 1986] C. Apte and S. J. Hong. Using Qualitative Reasoning to Understand Financial Arithmetic. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 942–948, Philadelphia, PA, August 1986.

[Bouwman, 1983] M. J. Bouwman. Human Diagnostic Reasoning by Computer: An Illustration from Financial Analysis. *Management Science*, 29(6):653–672, June 1983.

[Dague *et al.*, 1990] P. Dague, O. Jehl, and P. Taillibert. An Interval Propagation and Conflict Recognition Engine for Diagnosing Continuous Dynamic Systems. In *Proceedings of the International Workshop on Expert Systems in Engineering, in: Lecture Notes in AI*, Vienna, 1990. Springer.

[Davis, 1987] E. Davis. Constraint Propagation with Interval Labels. *Artificial Intelligence*, 32(3):281–332, July 1987.

[de Kleer and Williams, 1986] J. de Kleer and B. C. Williams. Back to Backtracking: Controlling the ATMS. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 910–917, Philadelphia, PA, August 1986.

[de Kleer, 1986a] J. de Kleer. An Assumption-Based TMS. *Artificial Intelligence*, 28(2):127–162, 1986.

[de Kleer, 1986b] J. de Kleer. Problem solving with the ATMS. *Artificial Intelligence*, 28(2):197–224, 1986.

[Dhar and Croker, 1988] V. Dhar and A. Croker. Knowledge-based Decision Support in a Business: Issues and a Solution. *IEEE Expert*, pages 53–62, Spring 1988.

[Dhar *et al.*, 1988] V. Dhar, B. Lewis, and J. Peters. A Knowledge-Based Model of Audit Risk. *AI Magazine*, 9(3):57–63, Fall 1988.

[Dressler and Farquhar, 1989] O. Dressler and A. Farquhar. Problem Solver Control over the ATMS. In *Proc. German Workshop on Artificial Intelligence*, 1989.

[Forbus and de Kleer, 1988] K. D. Forbus and J. de Kleer. Focusing the ATMS. In *Proc. 7th National Conf. on Artificial Intelligence*, pages 193–198, Minneapolis, MN, 1988.

[Hamscher, 1990] W. C. Hamscher. Explaining Unexpected Financial Results. In *Proc. AAAI Spring Symposium on Automated Abduction*, pages 96–100, March 1990. Available from the author.

[Lassez *et al.*, 1987] C. Lassez, K. McAloon, and R. Yap. Constraint Logic Programming and Option Trading. *IEEE Expert*, 2(3):42–50, Fall 1987.

[Peters, 1989] J. M. Peters. *A Knowledge Based Model of Inherent Audit Risk Assessment.* PhD thesis, Katz Graduate School of Business, University of Pittsburgh, 1989. Available from the author, School of Urban and Public Affairs, Carnegie Mellon University, Pittsburgh, PA 15213-3890.

[Reboh and Risch, 1986] R. Reboh and T. Risch. SYNTEL: Knowledge Programming Using Functional Representations. In *Proc. 5th National Conf. on Artificial Intelligence*, pages 1003–1007, Philadelphia, PA, August 1986.

[Simmons, 1986] R. G. Simmons. Commonsense Arithmetic Reasoning. In *Proc. 5th National Conf. on Artificial Intelligence*, Philadelphia, PA, August 1986.

[Sussman and Steele, 1980] G. J. Sussman and G. L. Steele. Constraints: A Language for Expressing Almost-hierarchical Descriptions. *Artificial Intelligence*, 14(1):1–40, January 1980.

[Williams, 1989] B. C. Williams. *Invention from First Principles via Topologies of Interaction.* PhD thesis, MIT, 1989.

A Methodology for Managing Hard Constraints in CLP Systems

Joxan Jaffar
IBM T.J. Watson Research Ctr.
P.O. Box 704
Yorktown Heights, NY 10598
(joxan@ibm.com)

Spiro Michaylov
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
(nch@cs.cmu.edu)

Roland Yap
IBM T.J. Watson Research Ctr.
P.O. Box 704
Yorktown Heights, NY 10598
(rhc@ibm.com)

Abstract

In constraint logic programming (CLP) systems, the standard technique for dealing with hard constraints is to delay solving them until additional constraints reduce them to a simpler form. For example, the CLP($\mathcal{R}$) system delays the solving of nonlinear equations until they become linear, when certain variables become ground. In a naive implementation, the overhead of delaying and awakening constraints could render a CLP system impractical.

In this paper, a framework is developed for the specification of *wakeup degrees* which indicate how far a hard constraint is from being awoken. This framework is then used to specify a runtime structure for the delaying and awakening of hard constraints. The primary implementation problem is the timely awakening of delayed constraints in the context of temporal backtracking, which requires changes to internal data structures be reversible. This problem is resolved efficiently in our structure.

1 Introduction

The Constraint Logic Programming scheme [7] prescribes the use of a constraint solver, over a specific structure, for determining the solvability of constraints. In practice, it is difficult to construct efficient solvers for most useful structures. A standard compromise approach has been to design a partial solver, that is, one that solves only a subclass of constraints, the *directly solvable* ones. The remaining constraints, the *hard* ones, are simply delayed from consideration when they are first encountered; a hard constraint is reconsidered only when the constraint store contains sufficient information to reduce it into a directly solvable form. In the real-arithmetic-based CLP($\mathcal{R}$) system [8, 9], for example, nonlinear arithmetic constraints are classified as hard constraints, and they are delayed until they become linear.

The key implementation issue is how to efficiently process just those delayed constraints that are affected as a result of a new input constraint. Specifically, the cost of processing a change to the current collection of delayed constraints should be related to the delayed constraints affected by the change, and not to all the delayed constraints. The following two items seem necessary to achieve this end.

First is a notion which indicates how far a delayed constraint is from being awoken. For

example, it is useful to distinguish the delayed CLP($\mathcal{R}$) constraint $X = max(Y, Z)$, which awaits the grounding of Y and Z, from the constraint $X = max(5, Z)$, which awaits the grounding of Z. This is because, in general, a delayed constraint is awoken by not one but a conjunction of several input constraints. When a subset of such input constraints has already been encountered, the runtime structure should relate the delayed constraint to just the remaining kind of constraints which will awaken it.

The other item is some data structure, call it the *access structure*, which allows immediate access to just the delayed constraints affected as the result of a new input constraint. The main challenge is how to maintain such a structure in the presence of backtracking. For example, if changes to the structure were trailed using some adaptation of PROLOG techniques [14], then a cost proportional to the number of entries can be incurred even though no delayed constraints are affected.

There are two main elements in this paper. First is a framework for the specification of *wakeup degrees* which indicate how far a hard constraint is from being awoken. Such a formalism makes explicit the various steps a CLP system takes in reducing a hard constraint into a directly solvable one. The second element is a runtime structure which involves a global stack representing the delayed constraints. This stack also contains all changes made to each delayed constraint when a new input constraint makes progress towards the awakening of the delayed constraint. A secondary data structure is the access structure. Dealing with backtracking is straightforward in the case of the global structure simply because it is a stack. For the access structure, no trailing/saving of entries is performed; instead, they are *reconstructed* upon backtracking. Such reconstruction requires a significant amount of interconnection between the global stack and access structure. In this runtime structure, the overhead cost of managing an operation on the delayed constraints is proportional to the size of the delayed constraints *affected* by the operation, as opposed to all the delayed constraints.

2 Background and Related Work

In this section we first review some early ideas of dataflow and local propagation, and the notion of flexible atom selection rules in logic programming systems. We then briefly review the basics of CLP, discuss the issue of delaying constraints in CLP, and mention some delay mechanisms in various CLP systems.

2.1 Data Flow and Local Propagation

The idea of dataflow computation, see e.g. [1], is perhaps the simplest form of a delay mechanism since program operations can be seen as directional constraints with fixed inputs and outputs. In its pure form, a dataflow graph is a specification of the data dependencies required by such an operation before it can proceed. Extensions, such as the I-structures of [2], are used to provide a delay mechanism for lazy functions and complex data structures such as arrays in the context of dataflow.

In local propagation, see e.g. [11], the solving of a constraint is delayed until enough of its variables have known values in order that the remaining values can be directly computed. Solving a constraint can then cause other constraints to have their values locally propagated, etc. The concept and its implementation are logical extensions of data flow – in essence, a data flow graph is one possible local propagation path through a constraint network. In other words, directionality is eliminated.

2.2 Delay Mechanisms in PROLOG

In PROLOG, the notion of delaying has been mainly applied to goals (procedure calls), and implemented by the use of a dynamically changeable atom selection rule. The main uses of delaying were to handle safe negation [10], and also to attempt to regain some of the completeness lost due to PROLOG's depth first search. There are similarities

between implementing delay in PROLOG and implementing a data flow system, except in one fundamental aspect: temporal backtracking[1]. Further complications are related to the saving of machine states while a woken goal is being executed.

Some PROLOGs allow the user to specify delay annotations. One kind is used on subgoals. For example, the annotation *freeze(X, G)* in PROLOG-II [4] defers the execution of the goal G until X is instantiated. Another kind of annotation is applied to relations. For example, the *wait* declarations of MU-PROLOG [10] and the *when* declarations of NU-PROLOG [13] cause all calls to a procedure to delay until some instantiation condition is met.

Carlsson [3] describes an implementation technique for *freeze(X, G)*. While an implementation of *freeze* can be used to implement more sophisticated annotations like *wait* and *when*, this will be at the expense of efficiency. This is mainly because the annotations can involve complicated conjunctive and disjunctive conditions. Since *freeze* takes just one variable as an argument, it is used in a complicated manner in order to simulate the behavior of a more complex wakeup condition.

In general, the present implementation techniques used for PROLOG systems have some common features:

- The delay mechanism relies on a modification of the unification algorithm [3]. This entails a minimal change to the underlying PROLOG engine. In CLP systems, this approach is not directly applicable since there is, in general, no notion of unification.
- Goals are woken by variable bindings. Each binding is easily detectable (during unification). In CLP systems, however, detecting when a delayed constraint should awaken is far more complicated in general. In this paper, this problem is addressed using wakeup degrees, described in the next section.
- The number of delayed goals is not large in general. This can render acceptable, implementations in which the cost of awakening a goal is related to the number of delayed goals [3], as opposed to the number of awakened goals. In a CLP system, the number of delayed constraints can be very large, and so such a cost is unacceptable.

[1]Committed choice logic programming languages [12] use delaying for process synchronization. However there is no backtracking here.

2.3 Delaying Hard Constraints in CLP

Before describing the notion of delaying constraints, we briefly recall some main elements of CLP. At the heart of a CLP language is a structure $\mathcal{D}$ which specifies the underlying domain of computation, the constant, function and constraint symbols, and the corresponding constants, functions and constraints. *Terms* are constructed using the constant and function symbols, and a *constraint* is constructed using a constraint symbol whose arguments are terms. An *atom* is a term of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol and the t_i are terms. A CLP *program* is a finite collection of *rules*, each of which is of the form

$$A_0 :- \alpha_1, \alpha_2, \ldots, \alpha_k$$

where each α_i, $1 \leq i \leq k$, is either a constraint or an atom, that is, a term of the form $p(t_1, \ldots, t_n)$ where p is a user-defined predicate symbol and the t_i are terms. The language CLP($\mathcal{R}$) for example, involves arithmetic terms, e.g. $X + 3 * Y + Z$ and constraints, e.g. $X + 3 * Y + Z \geq 0$.

The essence of the CLP operational model is that it starts with an empty collection CS_0 of constraints as its *constraint store*, and successively augments this store with a new *input* constraint. That is, each primitive step in the operational model obtains a new store CS_{i+1} by adding an input constraint to the previous store CS_i, $i \geq 0$. The conjunction of the constraints in each store is satisfiable. If every attempt to generate a collection CS_{i+1} from CS_i results in an unsatisfiable collection, then the store may be reset to some previous

store CS_j, $j < i$, that is, *backtracking* occurs. The full details of the operational model, not needed in this paper, can be obtained from [7].

In principle, a CLP system requires a decision procedure for determining whether a satisfiable constraint store can be augmented with an input constraint such that the resulting store is also satisfiable. In practice, this procedure can be prohibitively expensive. An incomplete system, but which is often still very useful, can be obtained by partitioning the class of constraints into the *directly solvable* ones, and the *hard* ones. Upon encountering a hard constraint, the system simply defers the consideration of this constraint until the store contains enough information to reduce the hard constraint into a directly solvable form[2].

There are a number of CLP systems with delayed constraints. One is PROLOG-II [4] where the hard constraints are disequations over terms, and these constraints awaken when their arguments become sufficiently instantiated. In CLP($\mathcal{R}$) , the hard constraints are the nonlinear arithmetic ones, and these delay until they become linear. In CHIP [6], hard constraints include those over natural numbers, and these awaken when both an upper and lower bound is known for at least all but one of the variables. In PROLOG-III [5], some hard constraints are word equations and these awaken when the lengths of all but the rightmost variables in the two constituent expressions become known.

3 Wakeup Systems

Presented here is a conceptual framework for the specification of operations for the delaying and awakening of constraints. We note that the formalism below is not designed just for logic programming systems.

Let the *meta-constants* be a new class of symbols, and hereafter, these symbols are denoted by α and β. A meta-constant is used as a template for a (regular) constant. Define that a *meta-constraint* is just like a constraint except that meta-constants may be written in place of constants. A meta-constraint is used as a template for a (regular) constraint.

To indicate how far a hard constraint is from being awoken, associate with each constraint symbol Ψ a finite number of *wakeup degrees*. Such a degree $\mathcal{D}$ is a template representing a collection of Ψ-constraints[3]. It is defined[4] to be either the special symbol *woken*, or a pair (t, C) where

- t is a term of the form $\Psi(t_1, \ldots, t_n)$ where each t_i is either a variable, constant or meta-constant, and
- C, is a conjunction of meta-constraints which contain no variables and whose meta-constants, if any, appear in t.

Let θ be a mapping from variables into variables and meta-constants into constants. An *instance* of a wakeup degree $\mathcal{D} = (t, C)$ is the constraint obtained by applying to t such a mapping θ which evaluates C into *true*. The instance is denoted $\mathcal{D}\theta$. In CLP($\mathcal{R}$) for example, a subset of the constraints involving the constraint symbol *pow* (where $pow(X, Y, Z)$ means $X = Y^Z$) may be represented by the degree $pow(A, B, \alpha)$, $\alpha \neq 0$. This subset contains all the constraints of the form $pow(X, Y, c)$ where X and Y are not necessarily distinct variables and c is a nonzero real number.

Associated with each wakeup degree $\mathcal{D}$ is a collection of pairs, each of which contains a *generic wakeup condition* and a wakeup degree called the *new degree*. Each wakeup condition is a conjunction of meta-constraints all of whose meta-constants appear in $\mathcal{D}$. Any variable in a wakeup condition which does not appear the in associated degree is called *existential*. An *instance* $\mathcal{W}\theta$ of a generic wakeup condition $\mathcal{W}$ is the constraint

[2] It is possible that hard constraints remained indefinitely deferred.

[3] These are constraints written using the symbol Ψ.

[4] This specific definition is but one way to represent a set of expressions. It may be adapted without affecting what follows.

obtained by applying the mapping θ which maps non-existential variables into variables and meta-constants into constants.

Like wakeup degrees, a wakeup condition represents a collection of constraints. Intuitively, a wakeup condition specifies when a hard constraint changes degree to the new degree. More precisely, suppose that $\mathcal{D}$ is a wakeup degree and that $\mathcal{W}$ is one of its wakeup conditions with the new degree $\mathcal{D}'$. Let C be a hard constraint in $\mathcal{D}$, that is, C is an instance $\mathcal{D}\theta$ of $\mathcal{D}$. Further suppose that the constraint store implies the corresponding instance of $\mathcal{W}$, that is, the store implies

$$\exists X_1 \ldots X_n(\mathcal{W}\theta)$$

where the X_i denote the existential variables of $\mathcal{W}$. Let C' denote a constraint equivalent to $C \wedge \exists X_1 \ldots X_n(\mathcal{W}\theta)$. We then say the constraint C *reduces* to the constraint C' *via* $\mathcal{W}$.

Consider once again the CLP($\mathcal{R}$) constraints involving *pow*. These constraints may be partitioned into classes represented by the wakeup degrees $pow(A,B,C)$, $pow(\alpha,B,C)$, $pow(A,\alpha,C)$, $pow(A,B,\alpha)$ and *woken*. For the degree $pow(A,B,C)$, which represents constraints of the form $pow(X,Y,Z)$ where X, Y and Z are variables, an example wakeup condition is $C = \alpha$. This indicates that when a constraint, e.g. $Z = 4$, is entailed by the constraint store, a delayed constraint such as $pow(X,Y,Z)$ is reduced to $pow(X,Y,4)$. This reduced constraint may have the new degree $pow(A,B,\alpha)$. Another example wakeup condition is $A = 1$, indicating that when a constraint such as $X = 1$ is entailed, a delayed constraint of the form $pow(X,Y,Z)$ can be reduced to $pow(1,Y,Z)$. This reduced constraint, which is in fact equivalent to the directly solvable constraint $Y = 1 \vee (Y \neq 0 \wedge Z = 0)$, may be in the degree *woken*. We exemplify some other uses of wakeup conditions below.

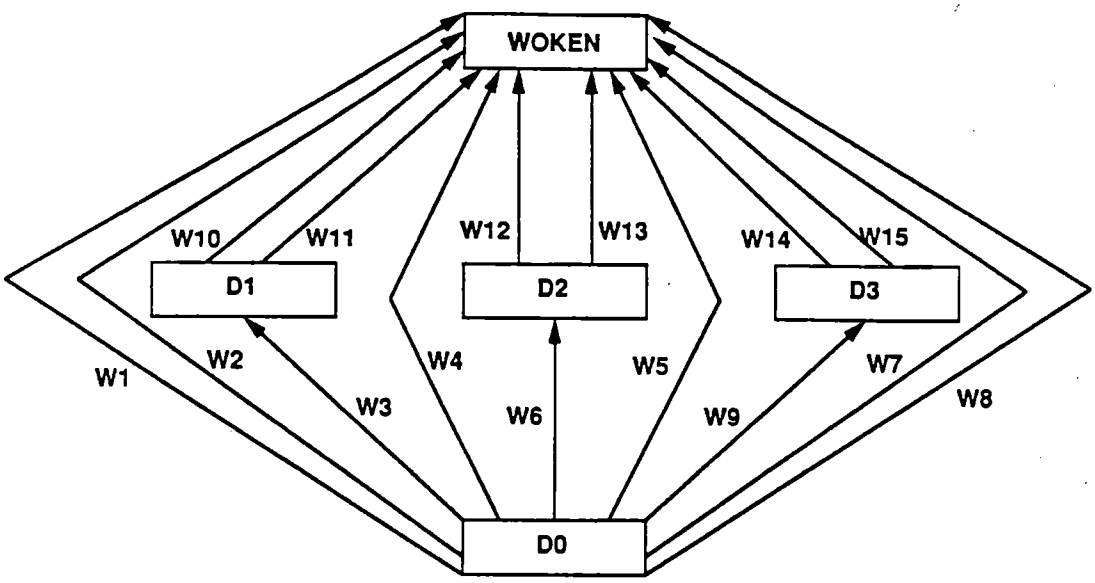
A *wakeup system* for a constraint symbol Ψ is a finite collection S of wakeup degrees for Ψ satisfying:

- S contains the special degree called *woken* which represents a subset of all the directly solvable Ψ-constraints.
- No two degrees in S contain the same Ψ-constraint.
- Let the Ψ-constraint C have a degree $\mathcal{D}$ which has a wakeup condition $\mathcal{W}$ and new degree $\mathcal{D}'$. Then every reduced constraint C' of C via $\mathcal{W}$ is contained in $\mathcal{D}'$.

The illustration in figure 1 contains an example wakeup system for the CLP($\mathcal{R}$) constraint symbol *pow*. A wakeup degree is represented by a node, a wakeup condition is represented by an edge label, and the new degree of a reduced constraint is represented by the target node of the edge labelled with the wakeup condition which caused the reduction.

Generic wakeup conditions can be used to specify the operation of many existing systems which delay constraints. In PROLOG-like systems whose constraints are over terms, awaiting the instantiation of a variable X to a ground term can be represented by the wakeup condition $X = \alpha$. Awaiting the instantiation of X to a term of the form $f(\ldots)$, on the other hand, can be represented by $X = f(Y)$ where Y is an existential variable. We now give some examples on arithmetic constraints. In PROLOG-III, for example, the wakeup condition $X \leq \alpha$ could specify that the length of word must be bounded from above before further processing of the word equation at hand. For CHIP, where combinatorial problems are the primary application, an example wakeup condition could be $\alpha \leq X \wedge X \leq \beta \wedge \beta - \alpha \leq 4$ which requires that X be bounded within a small range. For CLP($\mathcal{R}$) , an example wakeup condition could be $X = \alpha * Y + \beta$, which requires that a linear relationship hold between X and Y.

Summarizing, each constraint symbol in a CLP system which gives rise to hard constraints can be associated with a finite collection of wakeup degrees each of which indicate how far the constituent constraints are from being awoken. These degrees can be organized into a wakeup system, that is, a graph



Legend:

D0 : $pow(A, B, C)$ D1 : $pow(\alpha, B, C), \alpha \neq 0 \wedge \alpha \neq 1$
D2 : $pow(A, \alpha, C), \alpha \neq 0 \wedge \alpha \neq 1$ D3 : $pow(A, B, \alpha), \alpha \neq 0 \wedge \alpha \neq 1$

$w_1 : A = 0$	$w_2 : A = 1$	$w_3 : A = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$w_4 : B = 0$	$w_5 : B = 1$	$w_6 : B = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$w_7 : C = 0$	$w_8 : C = 1$	$w_9 : C = \alpha \wedge \alpha \neq 0 \wedge \alpha \neq 1$
$w_{10} : B = \alpha$	$w_{11} : C = \alpha$	$w_{12} : A = \alpha$
$w_{13} : C = \alpha$	$w_{14} : A = \alpha$	$w_{15} : B = \alpha$

Figure 1: Wakeup degrees for $pow/3$

whose nodes represent degrees and whose edges are represent the wakeup condition/new degree relation between two degrees. Such a wakeup system can be viewed as a deterministic transition system, and can be used to specify the organization of a constraint solver: the degrees discriminate among constraints so that the solver is able to treat them differently, while the wakeup conditions specify degree transitions of hard constraints with respect to new input constraints.

An important design criterion is that the entailed constraints corresponding to the wakeup conditions be efficiently recognizable by the constraint solver. This problem, being dependent on a specific solver, is not addressed in this paper. It is difficult in general[5]. In CLP($\mathcal{R}$) , for example, it is relatively inexpensive to perform a check if an equation like $X = 5$ is entailed whenever the constraint store is changed. The situation for an inequality like $X \leq 5$ is quite different.

[5]In PROLOG, this problem reduces to the easy check of whether a variable is bound.

4 The Runtime Structure

Here we present an implementational framework in the context of a given wakeup system. There are three major operations with hard constraints which correspond to the actions of delaying, awakening and backtracking:

1. adding a hard constraint to the collection of delayed constraints;
2. awakening delayed constraints as the result of inputting a directly solvable constraint, and

3. restoring the entire runtime structure to a previous state, that is, restoring the collection of delayed constraints to some earlier collection, and restoring all auxiliary structures accordingly.

The first of our two major structures is a stack[6] containing the delayed constraints. Thus implementing operation 1, delaying a hard constraint, simply requires a push on this stack. Additionally, the stack contains hard constraints which are reduced forms of constraints deeper in the stack. For example, if the hard constraint $pow(X, Y, Z)$ were in the stack, and if the input constraint $Y = 3$ were encountered, then the new hard constraint $pow(X, 3, Z)$ would be pushed, together with a pointer from the latter constraint to the former. In general, the collection of delayed constraints contained in the system is described by the sub-collection of stacked constraints which have no inbound pointers.

Figure 2 illustrates the stack after storing the hard constraint $pow(X, Y, Z)$, then storing $pow(Y, X, Y)$, and then encountering the entailed constraint $X = 5$. Note that this one equation caused the pushing of two more elements, these being the reduced forms of the original two. The top two constraints now represent the current collection of delayed constraints.

The stack operations can be more precisely described in terms of the degrees of the hard constraint at hand. This description is given during the definition of the access structure below.

Now consider operation 2. In order to implement this efficiently, it is necessary to have some access structure mapping an entailed constraint C to just those delayed constraints affected by C. Since there are in general an infinite number of entailed constraints, a finite classification of them is required. We define this classification below, but we assume that the constraint solver, having detected an entailed constraint, can provide access to precisely the classes of delayed constraints which change degree.

[6]Hereafter, the term stack refers to this structure.

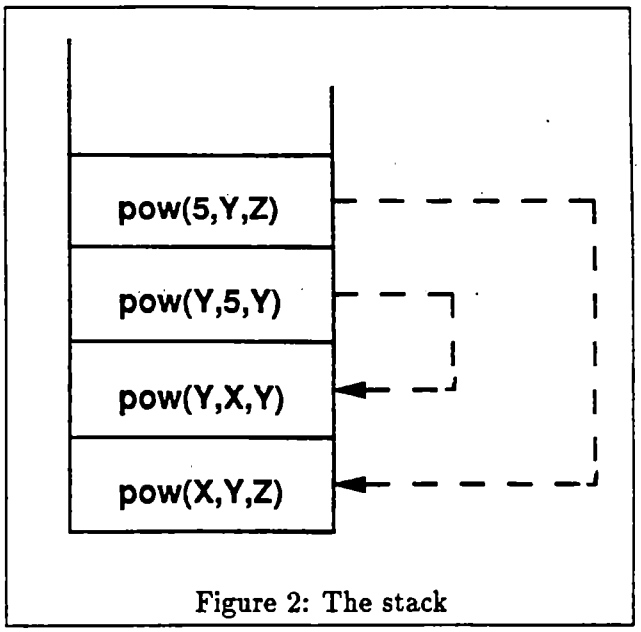


Figure 2: The stack

A *dynamic wakeup condition* is an instance of a generic wakeup condition $\mathcal{W}$ obtained by (a) renaming all the non-existential variables in $\mathcal{W}$ into runtime variables[7], and (b) instantiating any number of the meta-constants in $\mathcal{W}$ into constants. An *instance* of a dynamic wakeup condition is obtained by mapping all its meta-constants into constants.

A dynamic wakeup condition is used as a template for describing the collection of entailed constraints (its instances) which affect the same sub-collection of delayed constraints. For example, suppose that the only delayed constraint is $pow(5, Y, Z)$ whose degree is $pow(\alpha, B, C)$ with generic wakeup conditions $B = \alpha$ and $C = \alpha$. Then only two dynamic wakeup conditions need be considered: $Y = \alpha$ and $Z = \alpha$. In general, only the dynamic wakeup conditions whose non-existential variables appear in the stack need be considered.

We now specify an access structure which maps a dynamic wakeup condition into a doubly linked list of nodes. Each node contains a pointer to

[7]These are the variables the CLP system may encounter.

a stack element containing a delayed constraint[8]. Corresponding to each occurrence node is a reverse pointer from the stack element to the occurrence node. Call the list associated with a dynamic wakeup condition $\mathcal{DW}$ a *$\mathcal{DW}$-list*, and call each node in the list a *$\mathcal{DW}$-occurrence node*.

Initially the access structure is empty. The following specifies what is done for the basic operations. It is assumed, without loss of generality, that the variables in the wakeup system are disjoint from runtime variables, and that no existential variable appears in more than one generic wakeup condition.

4.1 Delaying a new hard constraint

To delay a new hard constraint C, first push a new stack element for C. Let $\mathcal{D}$ denote its wakeup degree and $\mathcal{W}_1, \ldots, \mathcal{W}_n$ denote the generic wakeup conditions of $\mathcal{D}$. Thus C is $\mathcal{D}\theta$ for some θ. Then:

- For each $\mathcal{W}_i$, compute the dynamic wakeup condition $\mathcal{DW}_i$ corresponding to C and $\mathcal{W}_i$, that is, $\mathcal{DW}_i$ is $\mathcal{W}_i\theta$.
- For each $\mathcal{DW}_i$, insert into the $\mathcal{DW}_i$-list of the access structure a new occurrence node pointing to the stack element C.
- Set up reverse pointers from C to the new occurrence nodes.

4.2 Processing an Entailed Constraint

Suppose there is a new entailed constraint, say $X = 5$. Then:

- Obtain the dynamic wakeup conditions in the access structure whose instances are implied by $X = 5$. If no such conditions exist (i.e. no delayed constraint is affected by $X = 5$ being entailed), nothing more needs to be done.
- Consider the lists L associated with the above conditions. Then consider in turn each delayed constraint pointed to by the occurrence nodes in L. For each such constraint C, perform the following.
  - Delete all occurrence nodes pointed to by C.
  - Construct the reduced form C' of C by replacing all occurrences of X by 5. (Recall that in general, C' is obtained by conjoining C with the entailed constraint.) Now push C' onto the stack, set up a pointer from C' to C, and then perform the modifications to the access structure as described above when a new delayed constraint is pushed.

Figure 3 illustrates the entire runtime structure after the two hard constraints $pow(X, Y, Z)$ and $pow(Y, X, Y)$ were stored, in this order. Figure 4 illustrates the structure after a new input constraint makes $X = 5$ entailed.

4.3 Backtracking

Restoring the stack during backtracking is easy because it only requires a series of pops. Restoring the access structure, however, is not so straightforward because no trailing/saving of the changes was performed. In more detail, the primitive operation of backtracking is the following:

- Pop the stack, and let C denote the constraint just popped.
- Delete all occurrence nodes pointed to by C.
- If there is no pointer from C (and so it was a hard constraint that was newly delayed) to another constraint deeper in the stack, then nothing more need be done.
- If there is a pointer from C to another constraint C' (and so C is the reduced form of

[8] The total number of occurrence nodes is generally larger than the number of delayed constraints.

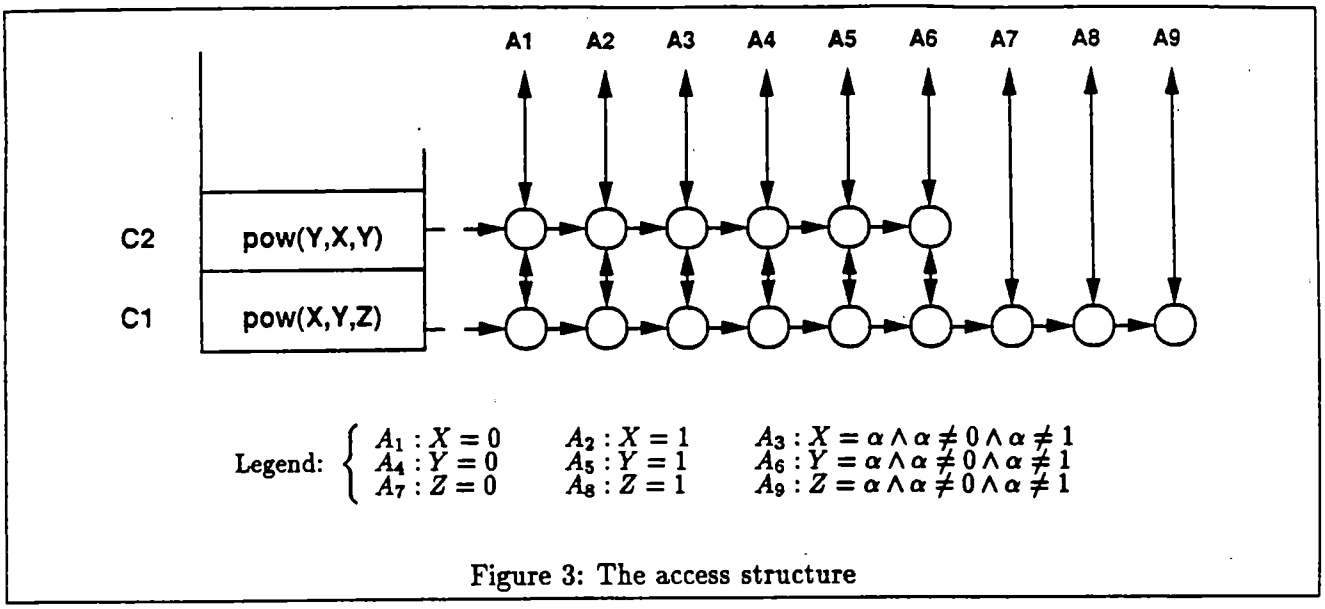


Figure 3: The access structure

C'), then perform the modifications to the access structure *as though* C' were being pushed onto the stack. These modifications, described above, involve computing the dynamic wakeup conditions pertinent to C', inserting occurrence nodes, and setting up reverse pointers.

Note that the access structure obtained in backtracking may not be structurally the same as that of the previous state. What is important, however, is that it depicts the same *logical* structure as that of the previous state.

4.4 Optimizations

Additional efficiency can be obtained by not creating a new stack element for a reduced constraint if there is no choice point (backtrack point) between the changed degrees in question. This saves space, saves pops, and makes updates to the access structure more efficient.

Another optimization is to save the sublist of occurrence nodes deleted as a result of changing the degree of a constraint. Upon backtracking, such sublists can be inserted into the access structure in constant time. This optimization, however, sacrifices space for time.

A third optimization is to merge $\mathcal{DW}$-lists. Let there be lists corresponding to the dynamic wakeup conditions $\mathcal{DW}_1, \ldots, \mathcal{DW}_n$. These lists can be merged into one list with the condition $\mathcal{DW}$ if

- the push of any delayed constraint C results in either (a) no change in any of the n lists, or (b) every list has a new occurrence node pointing to C;
- for every constraint C and for every mapping θ of meta-constants into constants, C implies $\mathcal{DW}\theta$ iff C implies $\mathcal{DW}_i\theta$ for some $1 \leq i \leq n$.

In the example of figure 3, the three lists involving X can be merged into one list which is associated with the dynamic wakeup conditions $X = \alpha$. Similarly for Y and Z.

4.5 Summary of the Runtime Structure

A stack is used to store delayed constraints and their reduced forms. An access structure maps a finite number of dynamic wakeup constraints to lists of delayed constraints. The constraint solver is assumed to identify those conditions for which an entailed constraint is an instance. The basic operations are then implemented as follows.

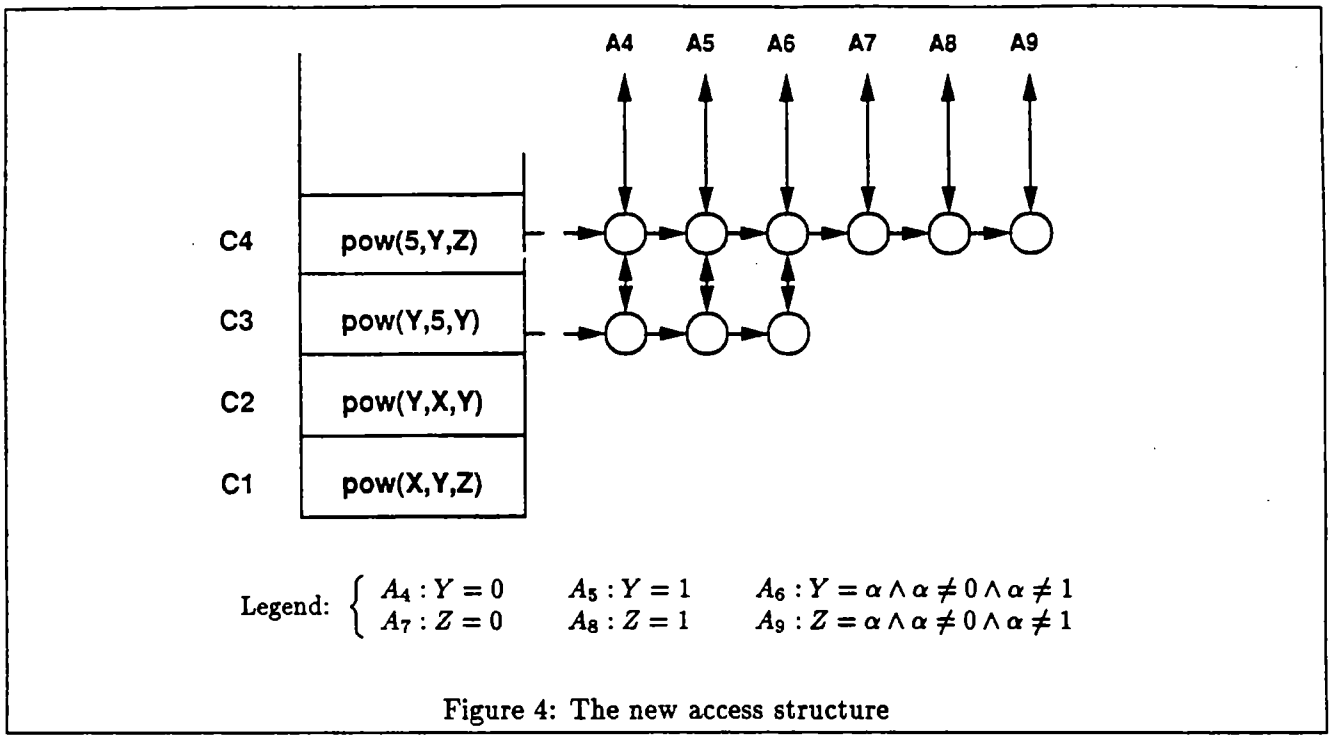


Figure 4: The new access structure

1. Adding a new constraint C simply involves a push on the stack, creating new occurrence nodes corresponding to C and the setting of pointers between the new stack and occurrence nodes. The cost here is bounded by the number of generic wakeup conditions associated with (the degree of) C.

2. Changing the degree of a constraint C involves a push of a new constraint C', deleting and inserting a number of occurrence nodes. Since the occurrence nodes are doubly-linked, each such insertion and deletion can be done in constant time. Therefore the total cost here is bounded by the number of generic wakeup conditions associated with C and C'.

3. Similarly, the cost in backtracking of popping a node C, which may be the reduced form of another constraint C', involves deleting and inserting a number of occurrence nodes. The cost here is again bounded by the number of generic wakeup conditions associated with C and C'.

In short, the cost of one primitive operation on delayed constraints (delaying a new hard constraint, upgrading the degree of one delayed constraint, including awakening the constraint, and undoing the delay/upgrade of one hard constraint) is bounded by the (fixed) size of the underlying wakeup system. The total cost of an operation (delaying a new hard constraint, processing an entailed constraint, backtracking) on delayed constraints is proportional to the size of the delayed constraints affected by the operation.

5 Concluding Discussion

A framework of wakeup degrees is developed to specify the organization of a constraint solver. These degrees represent the various different cases of a delayed constraint which should be treated differently for efficiency reasons. Associated with each degree is a number of wakeup conditions which specify when an input constraint changes the degree of a hard constraint. What is intended is that the wakeup conditions represent all the situations in which the constraint solver can efficiently update its knowledge about how far each delayed constraint is from being fully awoken.

The second part of this paper described a runtime structure for managing delayed constraints. A stack is used to represent the current collection of delayed constraints. It is organized so that it also records the chronological order of all changes made to this collection. These changes appear in the form of inserting a new delayed constraint, as well as changing the degree of an existing delayed constraint. An access structure is designed to quickly locate all delayed constraints affected by an entailed constraint. By an appropriate interconnection of pointers between the stack and the table, there is no need to save/trail changes made in the structure. Instead, a simple process of inserting or deleting nodes, and of redirecting pointers, is all that is required in the event of backtracking. By adopting this technique of performing minimal trailing, the speed of forward execution is enhanced, and space is saved, at the expense of some reconstruction in the event of backtracking. Even so, the overall overhead cost of the runtime structure for managing an operation to the delayed constraints is, in some sense, minimal.

Finally we remark that the implementation technique described has been used in the CLP($\mathcal{R}$) system for delaying hard constraints such as *multiply* and *pow*.

Acknowledgement. We thank Nevin Heintze and Michael Maher for their comments.

References

[1] Arvind and D.E. Culler, "Dataflow Architectures", in *Annual Reviews in Computer Science*, Vol. 1, Annual Reviews Inc., Palo Alto (1986), pp 225–253.

[2] Arvind, R.S. Nikhil and K.K. Pingali, "I-Structures: Data Structures for Parallel Computing", *ACM Transactions on Programming Languages and Systems*, 11(4) (October 1989) pp 598–632.

[3] M. Carlsson, "Freeze, Indexing and other Implementation Issues in the WAM", *Proceedings* 4^{th} *International Conference on Logic Programming*, MIT Press (June 1987), 40–58.

[4] A. Colmerauer, "PROLOG II Reference Manual & Theoretical Model," Internal Report, Groupe Intelligence Artificielle, Université Aix - Marseille II (October 1982).

[5] A. Colmerauer, "PROLOG-III Reference and Users Manual, Version 1.1", PrologIA, Marseilles (1990). [See also "Opening the PROLOG-III Universe", BYTE Magazine (August 1987).]

[6] M. Dincbas, P. Van Hentenryck, H. Simonis and A. Aggoun, "The Constraint Logic Programming Language CHIP", *Proceedings of the* 2^{nd} *International Conference on Fifth Generation Computer Systems*, Tokyo (November 1988), pp 249–264.

[7] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", *Proceedings* 14^{th} *ACM Symposium on Principles of Programming Languages*, Munich (January 1987), pp 111–119. [Full version: Technical Report 86/73, Dept. of Computer Science, Monash University, June 1986]

[8] J. Jaffar and S. Michaylov, "Methodology and Implementation of a CLP System", *Proceedings* 4^{th} *International Conference on Logic Programming*, MIT Press (June 1987), pp 196–218.

[9] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap, "The CLP($\mathcal{R}$) Language and System", IBM Research Report, RC 16292 (#72336), (November 1990).

[10] L. Naish, "Negation and Control in Prolog", Technical Report 85/12, Department of Computer Science, University of Melbourne (1985).

[11] G.L. Steele, "The Implementation and Definition of a Computer Programming Language Based on Constraints", Ph.D. Dissertation (MIT-AI TR 595), Dept. of Electrical Engineering and Computer Science, M.I.T.

[12] E.Y. Shapiro, "The Family of Concurrent Logic Programming Languages", *ACM Computing Surveys*, 21(3) (September 1989), pp 412–510.

[13] J.A. Thom and J. Zobel (Eds), "NU-PROLOG Reference Manual - Version 1.3" Technical Report 86/10, Dept. of Computer Science, University of Melbourne (1986) [revised in 1988].

[14] D.H.D. Warren, "An Abstract PROLOG Instruction Set", Technical note 309, AI Center, SRI International, Menlo Park (October 1983).

Comparing Solutions to Queries in Hierarchical Constraint Logic Programming Languages

Molly Wilson and Alan Borning
Department of Computer Science and Engineering, FR-35
University of Washington
Seattle, Washington 98195
molly@cs.washington.edu; borning@cs.washington.edu

1 Introduction

Hierarchical Constraint Logic Programming (HCLP) is an extension of Constraint Logic Programming (CLP) that allows a programmer to specify defaults and preferences, as well as required constraints. HCLP defines a family of languages that are parameterized both by the domain $\mathcal{D}$ over which the constraints are defined, and by the method, or *comparator*, that is used to select among potential solutions to the non-required constraints. In this paper, after a summary of the language, we describe various different comparators. We present a number of examples of using HCLP for scheduling and document formatting, and show how different comparators yield different solutions, with one comparator being more appropriate for one sort of problem, another comparator for another.

HCLP is based on the Constraint Logic Programming scheme [2, 9]. Several CLP languages have now been implemented, including CLP($\mathcal{R}$) [7, 10], Prolog III [3], CHIP [4, 8], and CLP(Σ^*) [11]. In our own papers on HCLP [1, 13], we present some of the theory of such languages, an algorithm for executing them, and a discussion of some of their nonmonotonic properties. To test our ideas, we also implemented two HCLP interpreters—the first one in CLP($\mathcal{R}$), and the second one in COMMON LISP. Recently, we have submitted a paper giving both a proof-theoretic and fixed point characterization of HCLP languages [12].

2 HCLP Programs

Rules in HCLP are of the form

$$p(\mathbf{t}) \text{ :- } q_1(\mathbf{t}), \ldots, q_m(\mathbf{t}), s_1 c_1(\mathbf{t}), \ldots, s_n c_n(\mathbf{t}).$$

where t denotes a list of terms, $p, q_1, \ldots, q_m$ are predicate symbols, $c_1, \ldots, c_n$ are constraints, and s_i indicates the strength of the corresponding constraint c_i. Symbolic names are given to the different strengths of constraints.

Operationally, goals are executed as in CLP, temporarily ignoring the non-required constraints, except to accumulate them. After a goal has been successfully reduced, there may still be non-ground variables in the solution. The accumulated hierarchy of non-required constraints is solved, using a method determined by the comparator, thus further refining the values of these variables. Additional answers may be produced by backtracking. As with CLP, constraints can be used multi-directionally, and the scheme can accommodate collections of constraints that cannot be solved by simple forward propagation methods.

Here is a sample HCLP($\mathcal{R}$) program that determines when an advisor and a student can meet. We strongly prefer that the advisor be free for at least the length of time of the meeting. This is not required, however, because other resources, such as a meeting room, may be more valuable than the professor's time. In addition, we prefer that the student have nothing else scheduled for that time, but given the normal pecking order, this preference is weaker than our desire that the advisor be available. (A post-revolutionary program, in which such class distinctions have been abolished, appears in Section 5.1.)

```
/* set up symbolic names for constraint strengths */
levels([required,strong,prefer]).

free(alan,9,11).
free(molly,10,12).

can_meet(StartTime,EndTime,Advisor,Student):-
     free(Advisor,StartAdvisor,EndAdvisor),
     strong StartAdvisor ≤ StartTime,
     strong EndAdvisor ≥ EndTime,
     free(Student,StartStudent,EndStudent),
     prefer StartStudent ≤ StartTime,
     prefer EndStudent ≥ EndTime.
```

To find an hour's meeting time for all Alan and Molly, we use the goal:

```
?- can_meet(S,E,alan,molly), required E - S = 1.
```

which succeeds with `S = 10` and `E = 11`.

3 Comparing Solutions

A *constraint hierarchy* is a multiset of labelled constraints. Given a constraint hierarchy H, H_0 denotes the required constraints in H with their labels removed. In the same way, we define $H_1, H_2, \ldots, H_n$ for levels $1, 2, \ldots, n$. We also define $H_k = \emptyset$ for $k > n$. A *solution* to a constraint hierarchy is a valuation for the free variables in the hierarchy, i.e., a function that maps the free variables of the constraints to elements in the domain $\mathcal{D}$ over which the constraints are defined. In the definition of a solution, we compare all valuations satisfying the required constraints, and reject those such that a "better" valuation exists.

The error function $e(c\theta)$ is used to indicate how nearly constraint c is satisfied for a valuation θ. This function returns a non-negative real number and must have the property that $e(c\theta) = 0$ if and only if $c\theta$ holds. ($c\theta$ denotes the result of applying the valuation θ to c.) For any domain $\mathcal{D}$, we can use the trivial error function that returns 0 if the constraint is satisfied and 1 if it is not. A comparator that uses this error function is a *predicate-only comparator*. For a domain that is a metric space, we can use its metric in computing the error instead of the trivial error function. (For example, the error for $X = Y$ would be the distance between X and Y.) Such a comparator is a *metric comparator*.

The error function $\mathbf{e}(C\theta)$ is a generalization of e to a list of constraints C. This error function returns a vector of individual constraint errors, one for each constraint in C.

Let g be a function that is applied to real-valued vectors and that returns some value that can be compared using <> and <. Then we can define the set of solutions to a constraint hierarchy H using the comparator

defined by g, $<>_g$, and $<_g$ as follows:

$$\begin{aligned} S_0 &= \{\theta \mid \forall c \in H_0\ e(c\theta) = 0\} \\ S &= \{\theta \mid \theta \in S_0 \ \wedge\ \forall \sigma \in S_0 \neg(g(\mathrm{e}(H_1\sigma)), \ldots, g(\mathrm{e}(H_n\sigma)) <_g g(\mathrm{e}(H_1\theta)), \ldots, g(\mathrm{e}(H_n\theta)))\} \end{aligned}$$

The $<_g$ relation for the error vectors in the above definition is defined in a manner similar to lexicographic order:

$$g(x_1, \ldots, x_n) <_g g(y_1, \ldots, y_n) \equiv \exists k \geq 1 \text{ such that } \forall i \in 1 \ldots k-1\ x_i <>_g y_i \ \wedge\ x_k <_g y_k$$

We now define g, $<>_g$, and $<_g$ for various comparators. (More intuitive descriptions of these comparators, in addition to the formal definitions, are given in [1, 6].) The weight for each constraint is denoted by w_i, where if v_i is the error for the ith constraint in a list, then w_i is the weight for that constraint. Each weight is a positive real number. For *weighted-sum-better*, *worst-case-better*, and *least-squares-better* g(**v**) returns a real number; for *locally-better* and *regionally-better* it returns a vector of real numbers.

For *weighted-sum-better*:

$$\begin{aligned} g(\mathbf{v}) &= \sum_{i=1}^{|\mathbf{v}|} w_i v_i \\ v <>_g u &\equiv v = u \\ v <_g u &\equiv v < u \end{aligned}$$

For *worst-case-better*:

$$\begin{aligned} g(\mathbf{v}) &= \max\{w_i v_i \mid 1 \leq i \leq |\mathbf{v}|\} \\ v <>_g u &\equiv v = u \\ v <_g u &\equiv v < u \end{aligned}$$

For *least-squares-better*:

$$\begin{aligned} g(\mathbf{v}) &= \sum_{i=1}^{|\mathbf{v}|} w_i v_i^2 \\ v <>_g u &\equiv v = u \\ v <_g u &\equiv v < u \end{aligned}$$

For *locally-better*:

$$\begin{aligned} g(\mathbf{v}) &= \mathbf{v} \\ \mathbf{v} <>_g \mathbf{u} &\equiv \forall i\ v_i = w_i \\ \mathbf{v} <_g \mathbf{u} &\equiv \forall i\ v_i \leq w_i \ \wedge\ \exists j \text{ such that } v_j < w_j \end{aligned}$$

For *regionally-better*:

$$\begin{aligned} g(\mathbf{v}) &= \mathbf{v} \\ \mathbf{v} <>_g \mathbf{u} &\equiv \neg(\mathbf{v} <_g \mathbf{u}) \wedge \neg(\mathbf{u} <_g \mathbf{v}) \\ \mathbf{v} <_g \mathbf{u} &\equiv \forall i\ v_i \leq w_i \ \wedge\ \exists j \text{ such that } v_j < w_j \end{aligned}$$

4 DeltaStar

The current HCLP($\mathcal{R}$) interpreter (the one written in COMMONLISP) uses an incremental constraint satisfaction algorithm called DELTASTAR[6, 5]. In HCLP, there are two situations in which an incremental algorithm can save computation. If backtracking occurs in the normal course of executing an HCLP program that contains multiple definitions of a predicate, the constraints arising from the old definition of the predicate must be retracted, and ones from the new definition added, but all other constraints are unaffected. An incremental algorithm allows solutions to be incrementally modified in this situation.

An incremental algorithm is also important for efficiency in connection with interactive graphics applications where answers must be produced before a goal is completely reduced—we cannot wait until all of the input events are known before computing display information. Instead, at appropriate points in program execution we need to solve the constraint hierarchy as generated up to that time. Again we'd prefer not to start from scratch in finding further solutions. The key in both of these situations is to develop algorithms that remove as few variable bindings as possible in attempting to resatisfy the hierarchy.

DELTASTAR is an algorithm for incrementally solving constraint hierarchies, but not for solving the constraints themselves. Rather, DELTASTAR is built above a *flat* incremental constraint solver which provides the actual constraint solving techniques (numeric, symbolic, iterative, local-propagation, etc.). Thus DELTASTAR is adaptable to many different constraint solving algorithms. The version of DELTASTAR that has been implemented in the current HCLP($\mathcal{R}$) interpreter uses variants of the Simplex algorithm to solve linear equality and inequality constraints. These variants provide the actual means of comparing solutions, but they know nothing about the hierarchy; this information is encapsulated in the DELTASTAR routine. This clean separation between the constraint solver and the hierarchy has allowed us to implement different comparators in HCLP($\mathcal{R}$) relatively quickly. By simply setting a variable, users can choose among the *weighted-sum-better*, *worst-case-better*, *locally-error-better*, and *regionally-error-better* comparators.

5 Applications and Examples

Having various comparators readily available has allowed us to begin experimenting with their suitability for certain domains, as well as for certain problems within a single domain. In this section, we present some examples that demonstrate the behavior of the different comparators, and make some preliminary observations on which ones are preferable for different classes of problems.

5.1 Scheduling

In Section 2 we presented a simple scheduling problem. Suppose we add the fact `free(bjorn,7,10).` to the earlier program and pose the query:

```
?- can_meet(S,E,alan,bjorn), can_meet(S,E,alan,molly), required E - S = 1.
```

If we are using the *locally-predicate-better* comparator, this query will succeed with two solutions: `S=10,E=11` and `S=9,E=10`. Both solutions satisfy Alan's preferences, whereas the first satisfies Molly's preferences while overriding Bjorn's and the second satisfies Bjorn's at the expense of Molly's. (In this example, multiple solutions arise not because of standard backtracking, but because there are multiple solutions to the constraint hierarchy that is created to satisfy the goal.)

If we apply the *locally-error-better* comparator, on the other hand, the solutions to the query are all hour-long intervals between 9 and 11. *Weighted-sum-better* and *regionally-error-better* also give these solutions.

Worst-case-better and *least-squares-better*, however, will select the hour between 9:30 and 10:30 as the best solution.

Now suppose that people's schedules change. The students also protest the professor's preferential status and change the meeting rule.

```
free(alan,7,8).
free(bjorn,8,9).
free(molly,11,12).

can_meet(StartTime,EndTime,Advisor,Student):-
     free(Advisor,StartAdvisor,EndAdvisor),
     prefer StartAdvisor ≤ StartTime,
     prefer EndAdvisor ≥ EndTime,
     free(Student,StartStudent,EndStudent),
     prefer StartStudent ≤ StartTime,
     prefer EndStudent ≥ EndTime.
```

If we again try to find a mutually acceptable meeting time for Alan, Bjorn, and Molly we find that *locally-predicate-better* yields three solutions — meeting for an hour starting at 7, 8, or 11. *Locally-error-better* yields an infinite number of solutions constrained to be any hour between 7 and 12. For this program, the *regional* comparators return the same solutions as their local counterparts. However, if we add a weaker constraint, for example one that weakly prefers meetings close to lunch time, the regional answers may be further refined and some of these solutions may be rejected. (For the local comparators, the set of solutions wouldn't be affected by this change.)

Weighted-sum-better selects the single meeting time at 8 in an attempt to "make the most people happy". *Worst-case-better* selects the single hour starting at 9 so that "no one person will be too put out".

We can conceive of scenarios where each of these solutions is most desirable. Normally, we might prefer to use a predicate comparator for scheduling meetings, so that we don't find ourselves meeting at strange times that are no good for anyone. Yet in some situations, such as deciding what time of year to meet, it is important to take exact error into account.

5.2 Document Formatting

The following is an example from the domain of document formatting. We want to lay out a table on a page in the most visually satisfying manner. We achieve this by allowing the white space between rows to be an elastic length. It must be greater than zero (or else the rows would merge together), yet we strongly prefer that it be less than 10 (because too much space between rows is visually unappealing). We do not want this latter constraint to be required, however, since there are some applications that may need this much blank space between lines of the table. We prefer that the table fit on a single page of 30 lines. Finally there is a default constraint that the white space be 5, that is if it is possible without violating any of the other constraints, and there is another constraint specifying the default type size.

```
levels([required,strong_prefer,prefer,default]).

table(PageLength, TypeSize,NumRow,WhiteSpace):-
     required (WhiteSpace + TypeSize) * NumRow = PageLength,
     required WhiteSpace > 0,
     strong_prefer WhiteSpace < 10,
```

```
prefer PageLength ≤ 30,
default WhiteSpace = 5.
default TypeSize = 11
```

If we use a predicate comparator, then if the **prefer** constraint cannot be satisfied and the table takes up more than one page, the **default** constraint will be satisfied, resulting in `WhiteSpace = 5`. However, if we use a metric comparator, spacing between the rows will be as small as possible to minimize the error in the `PageLength` constraint at the **prefer** level.

We can avoid this behavior by demoting the **prefer** constraint to a **default** so that the size of the type, the white space between rows, and the number of pages all interact at the same level in the hierarchy. *Weighted-sum-better* will characteristically choose the solution that minimizes the error for the majority of the constraints, while *worst-case-better* finds the middle ground.

6 Conclusion

We have shown how different comparators yield different solutions to sample Hierarchical Constraint Logic Programs for scheduling and page layout applications, and have argued that having this range of comparators is useful, since some solutions are more appropriate under some circumstances, other solutions under different circumstances. In the future we hope to investigate this point further using additional examples from finance and interactive graphics. In the financial examples, we would have preferential constraints regarding profit, risk, diversification, and so forth; as before, different comparators would select different solutions. In the interactive graphical examples, we would have preferences regarding window layout and dimensions, which again could be traded off in different ways by choosing different comparators.

Acknowledgements

This work was supported in part by the National Science Foundation under Grant No. IRI-8803294, by a gift from Apple Computer, and by a grant from the Washington Technology Center.

References

[1] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 149–164, Lisbon, June 1989.

[2] Jacques Cohen. Constraint Logic Programming Languages. *Communications of the ACM*, pages 52–68, July 1990.

[3] Alain Colmerauer. An Introduction to Prolog III. *Communications of the ACM*, pages 69–90, July 1990.

[4] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Bertheir. The Constraint Logic Programming Language CHIP. In *Proceedings FGCS-88*, 1988.

[5] Bjorn Freeman-Benson and Molly Wilson. DeltaStar: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies, October 1990. Submitted to the Eighth International Conference on Logic Programming.

[6] Bjorn Freeman-Benson and Molly Wilson. DeltaStar, How I Wonder What You Are: A General Algorithm for Incremental Satisfaction of Constraint Hierarchies. Technical Report 90-05-02, Department of Computer Science and Engineering, University of Washington, May 1990.

[7] N. Heintze, J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) Programmer's Manual. Technical report, Computer Science Dept, Monash University, 1987.

[8] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, MA, 1989.

[9] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich, January 1987.

[10] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Proceedings of the Fourth International Conference on Logic Programming*, pages 196–218, Melbourne, May 1987.

[11] Clifford Walinsky. CLP(Σ^*): Constraint Logic Programming with Regular Sets. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 181–196, Lisbon, June 1989.

[12] Molly Wilson. The Semantics of Hierarchical Constraint Logic Programming, October 1990. Submitted to the Eighth International Conference on Logic Programming.

[13] Molly Wilson and Alan Borning. Extending Hierarchical Constraint Logic Programming: Nonmonotonicity and Inter-Hierarchy Comparison. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, October 1989.

Coping With Nonlinearities in Constraint Logic Programming: Preliminary Results with CLPS($\mathcal{M}$)

Naim Abdullah*, Michael L. Epstein, Pierre Lim† and Edward H. Freeman
U S WEST Advanced Technologies
1545 Walnut Street
Boulder, CO 80302
naim@eecs.nwu.edu mepstein@uswest.com plim@uswest.com freeman@uswest.com

Extended Abstract

1 Introduction

Many "real world" problems require a mixture of numeric computation and symbolic reasoning. Constraint logic programming languages [1] have become recognized as a powerful tool in dealing with these problems. These languages provide symbolic reasoning capabilities via logical inferencing and numeric computation through an integrated constraint solver. Existing constraint logic programming languages such as CLP($\Re$) [2] and Prolog III [3] do not adequately address optimization problems nor do they try to solve nonlinear constraints.

Previously, we have reported on the integration of linear optimization with constraint logic programming in the design of the language CLPS($\mathcal{M}$) [4]. CLPS($\mathcal{M}$) allows natural specification of mixed integer linear optimization problems. Here we report on how CLPS($\mathcal{M}$) copes with nonlinear constraints. Since variables satisfying nonlinear constraints can be bound to complex numbers, CLPS($\mathcal{M}$) has the ability to deal with complex numbers as first class objects.

CLPS($\mathcal{M}$) is implemented as an instance of the Constraint Logic Programming Shell, CLPS [5]. CLPS is a domain independent constraint logic programming system that supports the connection of various domain specific "solvers". Within the CLPS framework, each solver is responsible for determining the satisfiability of the domain constraints encountered in the derivation path. If possible, the solver satisfies the system of constraints and obtains the bindings for as many of the variables in the constraints as it can. The solver also implements and exports the builtin predicates over the domain.

In the prototype for CLPS($\mathcal{M}$), we used a solver that utilizes Mathematica© [6] as the backend for satisficing some of the nonlinear constraints that are passed to it. Since Mathematica can solve a wide variety of nonlinear equations, CLPS($\mathcal{M}$) can solve many problems which the existing constraint logic languages are unable to solve. Section 2 gives some examples of the additional power available by having the ability to solve nonlinear constraints. Section 3 discusses in more detail how satisficing and optimization can be done in the presence of nonlinear constraints. Section 4 describes some of the open research issues raised by this work and section 5 gives our conclusions.

2 Examples

In this section, we illustrate through examples, the additional power available by having the ability to solve nonlinear constraints. As the examples show, the nonlinearities occur naturally in some problems, so having nonlinear constraint solving capabilities is important for these problems.

2.1 Circle

In Cohen's survey of constraint logic programming languages [7], the following is given as an example of the limitations of existing constraint logic languages:

```
on_circle(p(X, Y), c(A, B, (X - A)^2 + (Y - B)^2)).
```

The `on_circle` predicate states that a point `p(X, Y)` lies on the circumference of a circle centered at `(A, B)` with the square of the radius of the circle being `(X - A)^2 + (Y - B)^2` . The query,

*Graduate student at the Department of Computer Science, Northwestern University, Evanston, Illinois

†Student Associate – Intern in Science and Technology and a graduate student in the Department of Computer Science, Monash University, Clayton, Victoria, Australia 3168.

```
?- on_circle(p(7, 1), C), on_circle(p(0, 2), C).
```

specifies the family of circles C, passing through (7, 1) and (0, 2). The CLP($\Re$) interpreter is unable to solve this query and merely reformulates the constraints as:

```
Rsq = (7 - A) * (7 - A) + (1 - B) * (1 - B))
Rsq -  -A *  -A = (2 - B) * (2 - B)
```

and answers **Maybe** (**Rsq** denotes the anonymous variable used by CLP($\Re$) to represent the square of the radius of the circles in the family).

The same query in CLPS($\mathcal{M}$) returns:

```
Rsq = 625 - 350 * A + 50 * A ^ 2,
B = -23 + 7 * A
```

This is the result obtained by actually solving the nonlinear constraints that are generated rather than simply rearranging them. Cohen mentions a meta-interpreter at Brandeis University that is able to get the same result for this problem, but because we were concerned about performance on real world problems we decided to build a compiler-based system rather than pursue a meta-interpretation approach.

2.2 Complex Multiplication

One of the often repeated examples of the virtues of constraint logic programming languages is the complex multiplication program [8]. In CLPS($\mathcal{M}$), complex multiplication can be done by the builtin '*' predicate. We will discuss the program below for the insights it provides in nonlinear constraint solving:

```
zmul(c(R1, I1), c(R2, I2), c(R3, I3)) :-
     R3 = R1 * R2 - I1 * I2,
     I3 = R1 * I2 + R2 * I1.
```

It can be used for complex multiplication, if the first two arguments are instantiated as in the following query:

```
?- zmul(c(1, 1), c(2, 2), Z).
```

It can be used for complex division, if the third argument and either of the first two arguments are instantiated as in the following queries:

```
?- zmul(c(1, 1), Y, c(0, 4)).
?- zmul(X, c(2, 2), c(0, 4)).
```

Unfortunately, the constraint solvers of existing constraint logic programming languages are not powerful enough to permit this program to be used for finding the square roots of a complex number, as in the following query which attempts to find the square roots of 8i:

```
?- zmul(c(X,Y), c(X,Y), c(0,8)).
```

As is usual with nonlinearities, CLP($\Re$) answers **Maybe** and rearranges the constraints to:

```
X * X = Y * Y
8 - X * Y = X * Y
```

Given the same program, and this last query, CLPS($\mathcal{M}$) actually solves the system of nonlinear constraints to give a disjunction of all the possible solutions:

```
?- zmul(c(X,Y), c(X,Y), c(0,8)).

X = 2, Y = 2
 or
X = -2 * i, Y = 2 * i
 or
X = -2, Y = -2
 or
X = 2 * i, Y = -2 * i
```

The second solution is in fact equivalent to the third solution and the fourth solution is equivalent to the first solution. It as at this point that we realize that the program as given above for complex multiplication is deficient in that it fails to specify the constraints that in the complex number $X + iY$, X and Y are constrained to be real numbers. We use the builtin predicate **real/1** to enforce the constraint that a variable must be bound to a real value. This predicate succeeds if the argument is bound to a real value and fails if it is bound to something other than a real number. It delays if it is unbound. The program now becomes:

```
zmul(c(R1, I1), c(R2, I2), c(R3, I3)) :-
     R3 = R1 * R2 - I1 * I2,
     I3 = R1 * I2 + R2 * I1,
     real(R1), real(R2), real(R3),
     real(I1), real(I2), real(I3).
```

Now with the same query, CLPS($\mathcal{M}$) finds all the valid roots of 8i:

```
?- zmul(c(X,Y), c(X,Y), c(0,8)).

X = 2, Y = 2  or  X = -2, Y = -2 .
```

This example introduces the importance of having adequate capabilities in the language (such as the predicate **real/1**) in order to utilize the full power of the solver.

2.3 Circuits

Another example given in the CLP($\Re$) electrical engineering paper [8] is one of a simple circuit involving two parallel resistors.

```
resistor(V, I, R) :-
     V = I * R.

circuit(V, I1, I2, R1, R2, It) :-
     resistor(V, I1, R1),
     resistor(V, I2, R2),
     It = I1 + I2.
```

The `resistor` definition constrains a resistor to behave according to Ohm's law. The `circuit` definition places two resistors in parallel and constrains the total current in the circuit to be the sum of the currents passing through each. In many queries the pattern of instantiation of the variables permits CLP($\Re$) to find groundings for all the variables despite the fact that the constraint in `resistor/3` is nonlinear. For example

```
?- circuit(V, I1, I2, 10, 20, 50).

V = 333.333
I1 = 33.3333
I2 = 16.6667

*** Yes ***
```

Unfortunately, there are many situations when CLP($\Re$) must return with a **Maybe** and a set of constraints, when in fact the system is solvable. Consider how CLP($\Re$) copes with the query

```
?- circuit(V, 10, 50, It/R2, R2, It).

V = 50*R2
It = 60
It = 5*R2 * R2

*** Maybe ***
```

Now in CLPS($\mathcal{M}$) we once again extend the program to enforce the constraint that all resistances are positive:

```
circuit(V, I1, I2, R1, R2, It) :-
     resistor(V, I1, R1),
     resistor(V, I2, R2),
     It = I1 + I2,
     R1 > 0,
     R2 > 0.
```

Now the query

```
?- circuit(V, 10, 50, It/R2, R2, It).

It = 60.,
R2 = 3.46...,
V = 173.20...,
R1 = 17.32...
```

returns a grounding for all of the variables.

3 The CLPS($\mathcal{M}$) Solver

3.1 Architecture of the solver

An important design goal of CLPS($\mathcal{M}$) is that the new features that we are proposing like optimization and solving nonlinear constraints should not penalize the performance of those programs that do not use them. These programs should exhibit performance comparable to CLP($\Re$). Consequently, the prototype CLPS($\mathcal{M}$) solver contains a Gaussian subsolver, a Simplex subsolver and a nonlinear subsolver. The nonlinear subsolver is connected to Mathematica via 4.3 BSD UNIX^{TM} sockets. Programs that do not need the power of Mathematica need not pay the communication overhead of using it.

The nonlinear subsolver packages constraints as ASCII strings inside the Mathematica function `Reduce[ ]` and ships them to Mathematica. The answers are parsed by CLPS($\mathcal{M}$). Mathematica can return results that may be integer, rational, real or complex numbers. The parser in the nonlinear subsolver recognizes these four data types and converts them to real or complex numbers using a precision controllable by the programmer. The nonlinear subsolver eliminates multiple solutions that are redundant (for example, the two solutions to `X ^2 + 4 * X + 4 = 0` are reduced to one solution).

The bindings calculated by the subsolvers after processing a given constraint set are stored internally in solver data structures and variables are bound to these values by storing pointers in the value field of variable cells on the heap. After elimination of the redundant solutions, the solutions that remain are put in a "solver choicepoint"[5]. The solutions in a particular solver choicepoint are ordered in an arbitrary manner. The variables participating in the collected constraint set are bound to the first solution in the solver choicepoint. When a failure occurs later in the execution and the WAM backtracks to the current point, it is handed the next solution in the solver choicepoint.

The solver exports the predicates `real/1` and `complex/1` which succeed if their argument is bound to a real or a complex number respectively. The predicates fail if the arguments are bound to something other than a real or a complex number and they delay if the arguments are unbound. The solver also exports the

predicates **real/1** and **complex/1**. The arguments of these two predicates are bidirectional. The real and imaginary parts of a complex valued variable may be extracted by supplying the first argument. Or, a complex number may be determined by supplying the second argument to both the predicates.

3.2 Subsolver Selection Procedure

The constraint set passed to the solver will consist of a system of equations and/or a system of inequalities. If an objective function is present the solver will need to maximize or minimize the objective function subject to the constraints imposed by the system of equations and the inequalities. The coefficients of any of the variables in the objective function and in the system of equations and inequalities may be complex.

The Simplex algorithm works for real valued variables and objective functions. We do not know of any work in extending the Simplex algorithm to the complex domain. We conjecture that it will continue to work provided that certain precedences are observed amongst the variables in the objective function. We are working on proving this conjecture, but meanwhile our solver cannot solve constraint sets which would require Simplex to work on the complex domain. Systems of equations with complex coefficients are solved with the Gaussian subsolver.

When our solver is unable to solve a set of nonlinear constraints, it delays them until the instantiation of some variable participating in them. Then it considers them again to see if it can solve them given the additional information. This is analogous to the CLP($\Re$) strategy of delaying nonlinear constraints until they become linear. However, our strategy is more powerful in the sense that we do not wait for a constraint to become linear before trying to solve it again.

The programmer can perform optimization on a nonlinear collected constraint set by calling either of the following two predicates:

```
optMin(ObjFn?, ObjVal^, VarList?, SolnVect^)
optMax(ObjFn?, ObjVal^, VarList?, SolnVect^)
```

where ? refers to an input variable and ^ refers to an output variable.

The objective function and a list of variables appearing in it for which we want to find the bindings at the optimal solution, are passed in and the objective value and the solution vector are returned. Our solver does not yet have the capability to perform optimization when the objective function is nonlinear *at the time of execution*. Note that the objective function may initially be nonlinear and could become linear by the time it is encountered in the derivation path due to the instantiation of some variables appearing in it.

If the inequalities in the system of constraints are nonlinear our current prototype cannot perform optimization. This could actually be done by introducing slack variables and converting the nonlinear inequalities to nonlinear equations. Solving the resulting nonlinear system of equations would yield values for the slack variables. The values of the slack variables will indicate whether the original nonlinear inequalities are satisfiable. We have not implemented this in the prototype because we have not yet decided if the benefits are worth the costs.

The system of equations in the collected constraint set can include some nonlinear equations. In these cases the solver will consult Mathematica for the solutions to the nonlinear equations. Some of the solutions of the nonlinear equations may be complex (with nonzero imaginary parts). It will then repeatedly substitute the solutions in the system of equations and derive a set of of linear systems. Simplex is then run on the linear systems having real coefficients and the best values amongst them for the objective functions is chosen. The (real) linear system giving the best value for the objective function is then stacked with the complex linear systems on a solver choicepoint. The solver gives back the best value for the objective function found amongst the real linear systems, and upon backtracking hands out the complex linear systems of constraints (on the solver choicepoint) along with a **Maybe** answer.

A detailed algorithm is given below for the subsolver selection done by the solver:

```
if it is an optimization problem with a
  nonlinear_objective_function then
    /* Special purpose algorithms needed which are
       not yet implemented. */
    return Maybe
else if any nonlinear_inequalities in
  constraint set then
    /* Could introduce slack variables but not
       yet implemented. */
    remove the nonlinear_inequalities from the
      constraint set and delay them, restart the
      subsolver selection procedure with the
      modified constraint set
else if have objective_function/inequalities with
  complex coeffs or (have objective_function/
  inequalities with real coeffs and the system
  of equations has at least one complex coeff) then
    /* Note: A constant appearing alone is also
       considered a ''coeff''. Need complex Simplex
       for this which is not yet implemented. */
    return Maybe
else if linear system of equations with no
  objective_function and no inequalities then
    /* Sys of equations may have complex coeffs
```

```
      so can't use Simplex. */
    run Gaussian subsolver
else if constraints are linear then
    /* Everything real and have linear equations
       and/or linear inequalities and/or
       an objective function. */
    run Simplex
else if it is an optimization problem then
    /* Sys of equations is nonlinear with real
       coeffs. The inequalities, if present, are
       real and linear. The objective_function is
       real and linear. */
    solve the subset of nonlinear equations in the
      system of equations by Mathematica
    if Mathematica cannot solve them then
        return Maybe
    end if
    /* run Simplex on all the real branches of the
       solutions and get the best value */
    for all the real branches in the solution
      from Mathematica
        substitute the current solution in the
          rest of the equations
        run Simplex on the linear system
          constructed, and calculate the value
          of the objective function
        update the best value found so far for the
          objective function
    end for
    stack the best value found in the real branches
      with the complex branches found by Mathematica
      in a solver choicepoint
else
    /* System of equations is nonlinear with possibly
       complex coeffs. */
    solve the system of equations by Mathematica
    if Mathematica cannot solve them then
        remove the subset of nonlinear equations from
          the constraint set and delay them, restart
          the subsolver selection procedure with the
          modified constraint set
    end if
    arbitrarily order the solutions and push them
      in a solver choicepoint
end if
```

4 Future Research

Many interesting research issues are raised in the context of this work. As mentioned earlier, a Simplex algorithm working in the complex domain would make our solver more powerful. Such an algorithm would be useful in it's own right and would find applications in Electrical Engineering where currents are modeled as complex quantities.

We also need to resolve the issue of whether it is always useful to solve a nonlinearity immediately versus delaying it for sometime. The conditions under which it is beneficial to delay nonlinearities should be investigated.

The order in which the nonlinearities are solved and the order in which the solutions of a nonlinear equation are put in the solver choicepoint, also needs a more detailed examination.

5 Conclusion

Nonlinear constraints occur naturally in many problems. These problems are difficult or impossible for existing constraint logic programming languages. By having nonlinear constraint solving abilities in the solver, CLPS($\mathcal{M}$) has widened the scope of problems that can be attacked by constraint logic programming languages without compromising on performance for those problems that do not need these features.

References

[1] J. Jaffar and J-L. Lassez, "Constraint Logic Programming", *Procs. POPL 87*, Munich, January 1987.

[2] J. Jaffar and S. Michaylov, "Methodology and Implementation of a CLP System", Proceedings of the 4^{th} International Conference on Logic Programming, Melbourne, 1987, pp. 196–218.

[3] A. Colmerauer, "Final Specifications for PROLOG -III", Manuscript: ESPRIT Reference Number P1219(1160), February 1988. (See also "Opening the PROLOG-III Universe", BYTE Magazine, August 1987.)

[4] P. Lim, M. Epstein, E. H. Freeman, "A Constraint Logic Programming Language for Combinatorial Optimization and Linear Programming," accepted for publication in 7^{th} IEEE Conf. on Artificial Intelligence Applications, 1991.

[5] P. Lim and P. Stuckey, "A Constraint Logic Programming Shell", Proceedings of the 1990 Workshop on Programming Language Implementation and Logic Programming, Linköping, Sweden, August 20–22, Springer–Verlag, 1990.

[6] S. Wolfram, *Mathematica: A System for Doing Mathematics by Computer*, Addison-Wesley, U.S.A., 1988.

[7] J. Cohen, "Constraint Logic Programming Languages," CACM, Vol. 33, No. 7, July 1990.

[8] N. Heintze, S. Michaylov and P. Stuckey, "CLP($\Re$) and Some Electrical Engineering Problems", Proceedings of the 4^{th} International Conference on Logic Programming, Melbourne, 1987, pp. 675–703.

Dataflow Dependency Backtracking in a New CLP Language

William S. Havens

Expert Systems Laboratory
Centre for Systems Science
Simon Fraser University
Burnaby, British Columbia
CANADA V5A 1S6
email: havens@cs.sfu.ca

Abstract

Constraint-based reasoning and logic programming have been recently joined in the new field of constraint logic programming (CLP). There is keen research interest and new CLP languages are emerging. Intelligent backtracking techniques have also been extensively explored and adapted to logic programming. As well, object-oriented knowledge structures have been incorporated into the logic programming paradigm. All three of these capabilities are important for model-based hypothetical reasoning systems. We have been developing a new CLP language called Echidna which supports model-based reasoning using object-oriented knowledge structures; a clausal reasoner based on SLD-resolution; constraint reasoning for discrete, integer and real variables; and complete dependency backtracking using a justification-type reason maintenance system (RMS). In this short paper, we describe the integration of dependency backtracking into Echidna. We note some difficulties encountered with implementing dependency backtracking in CLP languages especially with persistent object variables. A new methodology called dataflow dependency backtracking is developed which overcomes these problems.

1. Introduction

Rapid developments are being made in the new field of *constraint logic programming* (CLP) as characterized by Jaffar and Lassez [21]. CLP inherits the methodologies of logic programming and constraint reasoning, replacing unification driven search with constraint propagation and constraint solving whenever possible. New CLP languages incorporating various constraint reasoning techniques continue to appear including PrologIII [4], CHIP [26] and recently Echidna [18].

Research in constraint reasoning methods for solving constraint satisfaction problems (CSPs) has a long history including [28, 22, 10]. Consistency techniques can be used to enhance backtrack search via *look-ahead* schemes [13, 26] and *look-back* schemes [24, 6]. The *backjumping* method of [11] is a look-back scheme central to intelligent backtracking.

Intelligent or dependency backtracking [25] is being actively explored for logic programming. Given a particular failure, dependency backtracking attempts to identify the actual goal (called the *culprit*) which caused the failure. Backtracking to the culprit is potentially much more efficient than blind chronological backtracking. Static *data dependency analysis* can identify the possible causes of a failure

[2, 20]. *Unification analysis* can pinpoint the culprit more accurately but with an associated execution time overhead [1, 9, 27]. The latter approach usually involves a *reason maintenance system*[1] (RMS) [7, 8] to record which goal choices have led to the failure. One of these choices (usually the most recent to maintain completeness) is identified as the culprit for backtracking. If the RMS retains the set of choices known to fail (called a *nogood*) in a nogood database, then that particular goal *environment* need never be attempted again. Two major types of RMSs are employed: the justification-type [8] maintains a single reasoning context which is explored sequentially; while assumption-based systems [7] manage multiple consistent environments. The simpler justification RMS is appropriate for dependency backtracking in logic programming languages [9] and can be extended to CLP reasoners as well.

Structured object-centered knowledge representations are important for model-based reasoning systems [5, 12, 23, 17]. Current Horn clause programming languages provide no significant knowledge structuring capabilities. To remedy this situation, object-oriented principles are being extended to logic programming [30, 3].

We have been developing a new CLP language called *Echidna* which supports model-based reasoning using object-oriented knowledge structures; a clausal reasoner based on SLD-resolution; constraint reasoning for discrete, integer and real variables; and complete dependency backtracking using a justification-type RMS. The goal of our research is a synthesis of these techniques into a simple coherent reasoning architecture. A major research issue has been incorporation of intelligent backtracking into the underlying CLP language. This problem is exacerbated by the inclusion of persistent logical variables associated with objects. We have overcome the problem by introducing a technique we call *dataflow dependency backtracking*.

Echidna is most similar to CHIP [26] for discrete and integer variables but with an integral object-oriented knowledge representation and an efficient dependency backtracking control structure. The architecture is based on the logical dataflow of hypothetical information among unified clauses and objects. Extensions in Echidna include dependency backtracking applied to clause unification, constraint propagation and object message passing. Like Drakos[9], we discard dependency information on failure thereby limiting the size of the nogood database. Furthermore, we reuse existing clauses in the proof tree whenever possible thereby significantly improving the efficiency of the backtrack search. A long paper describing these optimizations is in preparation [14].

The Echidna language is intended as a next generation expert systems programming environment for application to real-time, model-based reasoning tasks. A preliminary version of Echidna has been successfully implemented and is being applied to intelligent process control [15], automobile diagnosis [16] and intelligent CAD. A commercial version of the software and programming environment is under development [19].

In this short paper, we describe some problems of intelligent backtracking in CLP languages and the additional problems introduced by persistent logical variables. We introduce the notion of hypothetical dataflow of information among unified clauses and objects. Finally, we describe the dependency backtracking scheme.

[1] Also known as *truth maintenance systems*.

2. Intelligent Backtracking

This section examines some of the issues in dependency backtracking. Some of the assumptions appropriate for logic programming are less viable for CLP. When persistent logical variables are introduced, the problems get worse. We conclude that a new view of dependency backtracking is required.

2.1 Reuse of Existing Subgoals

How realistic are the assumptions behind dependency backtracking? Is the additional overhead required for correctly identifying the culprit on failure worth it? Empirical results suggest maybe [9, 27, 1]. We argue that for CLP languages, it is both more difficult and worth it. Consider the following examples. We assume Prolog's left-to-right computation rule. Given the goal clause (1) below, suppose that p_1 and p_2 succeed on particular values for their arguments X and Y, but p_3 fails on this value for Y:

(1) $\leftarrow p_1(Y), p_2(X), p_3(Y).$

What is the backtrack point? Chronological backtracking assumes that every previous goal instantiation may have caused the failure thereby choosing $p_2(X)$ as the culprit. Unfortunately, goal p_2 has no possible influence on variable Y so inevitably p_3 will generate the *identical* failure again. What is worse, all of the possible solutions for p_2 will be eventually tried, each causing p_3 to again generate the same repeated failure (called *thrashing* [22]). Dependency backtracking will correctly identify $p_1(Y)$ as the proper culprit. Goal p_1 has most recently (in the chronological order) had a *possible* effect on Y. Then all succeeding goals (in the ordering) are undone and the culprit retried for a different instantiation for Y. The method maintains completeness because no instantiation of $p_2(X)$ in the conjunctive subexpression of (1):

(2) $p_2(X), p_3(Y)$

can succeed (as argued above). Consequently, the search tree for these two goals can be deleted from consideration *in toto* without missing any possible solutions of the original goal (1).

However, the method is not optimal. Further significant improvements in dependency backtracking are possible. Consider again expression (1). It is clear that goal $p_2(X)$ cannot be the culprit for the failure of $p_3(Y)$ but it should also be obvious that neither can $p_1(Y)$ be a future culprit for any failure of $p_2(X)$. They share no common variables being independent subgoals in the expression. Consequently, it makes no sense to undo the current instantiation of p_2 when p_1 is identified as the proper culprit for dependency backtracking (as above). Clearly the instantiation for $p_2(X)$ is already correct. It need not be undone and retried from scratch for all possible clauses which could satisfy its goal. In particular, every instantiation for X which has already been rejected by p_2 would also be rejected if the goal were restarted. As well, every future possible instantiation for X which has not been explored would not yet

be explored if p_2 were restarted. Inevitably, p_2 would return to the exact state it attained at the time of the backtracking failure of p_3.

We conclude for the common occurrence of independent subgoals that dependency backtracking should differentiate between identifying the culprit and selecting which subsequent subgoals need to be undone. They are separate issues.

Having separated the identification of the culprit from the necessary undoing of dependent subgoals, we can further improve the efficiency of dependency backtracking. Consider a more complex version of the goal of (1) which shares variables between the culprit p_1 and subgoals p_2 and p_3 as goal (3) below:

(3) $\leftarrow p_1(X, Y), p_2(X), p_3(Y)$

Now both p_1 and p_2 are mutually dependent subgoals because they share a common logical variable X. Likewise, so are p_1 and p_3 which share variable Y. If we evaluate (3) from left to right, then the instantiation of p_2 can depend (but not necessarily[2]) on the particular instantiation chosen for p1. However, we can still improve dependency backtracking even when subgoals are not independent.

Consider the simplest case. After backtracking to $p_1(X, Y)$ for another instantiation, suppose it succeeds with a new value for Y but the value for X remains unchanged. The issue is whether the dependent subgoal p_2 need be undone and restarted with the new instantiation for p_1. Of course, it need not because its argument X has not changed. So the existing instantiation of p_2 is known to be correct even though p_1 and p_2 are mutually dependent subgoals. No doubt other implementations of dependency backtracking do not incorporate this improvement because it necessitates a more complex storage allocation scheme than the popular WAM [29] architecture provides. However, we must abandon the strict stack architecture of WAM anyway (as is argued in §2.4 below).

2.2 Bidirectional Dataflow

CLP introduces new complications for dependency backtracking. Information can propagate in both directions within a clause in CLP languages. Logical data can flow from any instantiated goal to all other goals in a conjunct whatever their particular textual or chronological order. Constraint propagation among related goals sharing common variables is bidirectional by nature. Whenever a variable is refined by one goal, all sister goals sharing that variable must incorporate the updated information.

Consider the following definite clause:

(4) $q(X, Y) :- p_4(X, Y), p_5(X), p_6(X, Y).$

where X and Y are both domain variables[26] and the goals, p_4, p_5 and p_6, are all constraints.[3] Domain variables are refined under constraint propagation by reducing their domains monotonically. We denote a change to domain variable X as X' and write conveniently $X' \subseteq X$ to mean that the domain D_X

[2]Static data dependency analysis would not be able to distinguish this situation whereas unification analysis could.

[3]Constraints in Echidna include primitive constraints and definite clauses with embedded constraints.

of variable X has been refined to D'_X such that $D'_X \subseteq D_X$. The *binding* of a domain variable is its currently associated domain.

Suppose constraint p_5 has been applied to X yielding a refinement X' which needs to be propagated to constraints p_4 and p_6 which share variable X. There may be some values in variable Y which are no longer consistent with the new X'. So X' must be propagated in both directions as indicated in Figure 1 below. If either p_4 or p_6 refine variable Y to Y', then consistency must also be propagated along the dataflow path for Y to the other constraint.

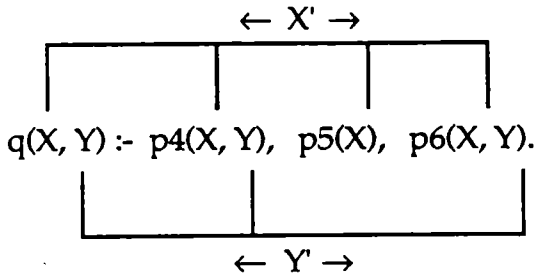


Figure 1: Bidirectional Dataflow in Definite Clauses

The bidirectional flow of data in CLP languages complicates the implementation of efficient dependency backtracking. Data can flow both ways in clauses during backtracking as well as during constraint propagation. In contrast, stack-based Prolog architectures [29] make a number of simplifying assumptions which are not appropriate here:

- Every node in the proof tree preceding the culprit (in chronological order of elaboration) is unaffected by the retraction of the culprit goal. Only clauses which can successfully unify with the existing variable bindings of these nodes will be tried for the culprit goal. This is not true for dependency backtracking in CLP reasoners since constraints can have been propagated in both directions. Retrying the culprit goal with another clause which unifies with the preceding variable environment may still require constraints to be propagated into this environment.
- Every node in the proof tree following and including the culprit goal is to be undone and retried from scratch. This strategy is easy to implement in a stack discipline but is very inefficient. As argued previously, there may be significant proof subtrees following the culprit node which either: 1) do not share any variables with the culprit goal; or 2) if they do share common variables, their new bindings are consistent with the current instantiations for these nodes. A forward-looking data dependency analysis [2, 20] could detect the first case but a dynamic dataflow analysis is required for the latter.

For example, please reconsider the expression of (4) and the dataflow diagram of Figure 1. Suppose the culprit is identified as $p_5(X)$ and a new instantiation for the goal is attempted producing a new binding X'. We note that X and X' may have any arbitrary relationship. Consequently, the backtracking of p_5 introduces a bidirectional dataflow propagation of X' which is monotonic to p_4 but may be nonmonotonic to p_6. In the case of p_4, it is necessary to apply constraint propagation to the updated value X'. For a nonmonotonic case on p_6, we must attempt to reunify its current clausal instantiation

with the new goal $p_6(X', Y)$. If p_6 is realized as a definite clause, then it may be necessary to repeat the dataflow of nonmonotonic information into the goals of its body. By so doing, we can determine which branches of the existing proof tree for $p_6(X, Y)$ can be reused for $p_6(X', Y)$ and which branches need to be undone and their goals restarted. Other schemes for dependency backtracking would always undo $p_6(X, Y)$ and restart $p_6(X', Y)$ anew regardless of how much of their derivation is common.[4]

2.3 Nonlocal Data Dependencies

We return to the important issue of the nonlocal side-effects of logical variable propagation in CLP languages. Unlike chronological backtracking, the culprit goal may occur anywhere in the proof tree and consequently backtracking propagated arbitrarily far in the derivation. In the definite clause of (4), suppose again that $p_5(X)$ is backtracked yielding a new binding X'. Besides propagating this new information to nodes p_4 and p_6 inside the clause as described above, we must also propagate X' outside the clause to the variable U of goal q(U, V) which was unified with (4), to every sister goal of q(U, V) which shares variable U, and if q(U, V) appears in the body of a definite clause whose arguments include U, then the dataflow backtracking must be propagated outside that clause, and so on. Figure 1 illustrates the dataflow paths within the clause.

2.4 Persistent Object Variables

Persistent object variables present additional difficulties. A *schema* in Echidna is a representation for a class of objects in the normal object-oriented tradition. Schemata contain persistent object variables called parameters and collections of logical methods for manipulating these variables. A schema instance is a valid term in the logic which may be created, bound to a variable, unified with other instances as well as be sent messages. From the logic programming perspective, the schema is a generative representation for a relation over its parameters. The logical methods defined within the schema are definite clauses which manipulate these variables. A particular relation is obtained by sending message goals to the schema which generatively construct a constraint network representing the relation.[5]

The variables of a schema instance reside within that instance and hence persist beyond the elaboration of any particular clause defined as a method within the schema. This property is essential for recording object state within the instance. Consequently, the RMS must keep track for each persistent variable the history of derivations which have influenced its binding. These derivations may be the result of the multiple and distinct goal elaborations. Any of the goals involved may be the proper culprit for dependency backtracking.

Figure 2a contains an Echidna definition of a schema called *foo*. The schema has two persistent logical variables (parameters). The parameter X is a real variable while the parameter Y has a discrete domain of colours. There is also a third persistent Herbrand local variable Z. Within the schema there are three logical methods defined: r(Z), q(X, Y) and colour(Y). These methods are accessed by sending logical messages to the schema using the *send* operator below:

[4]We present more details of reusing existing proof subtrees in [Havens91].

[5]More details of object programming in Echidna can be found in [HavSide90].

S:m(X, ...)

where S is the receiving schema; m is the message; and (X, ...) are the message arguments. For example, to assign the colour of some instance F1 of foo, we could send the message,

F1:colour(red).

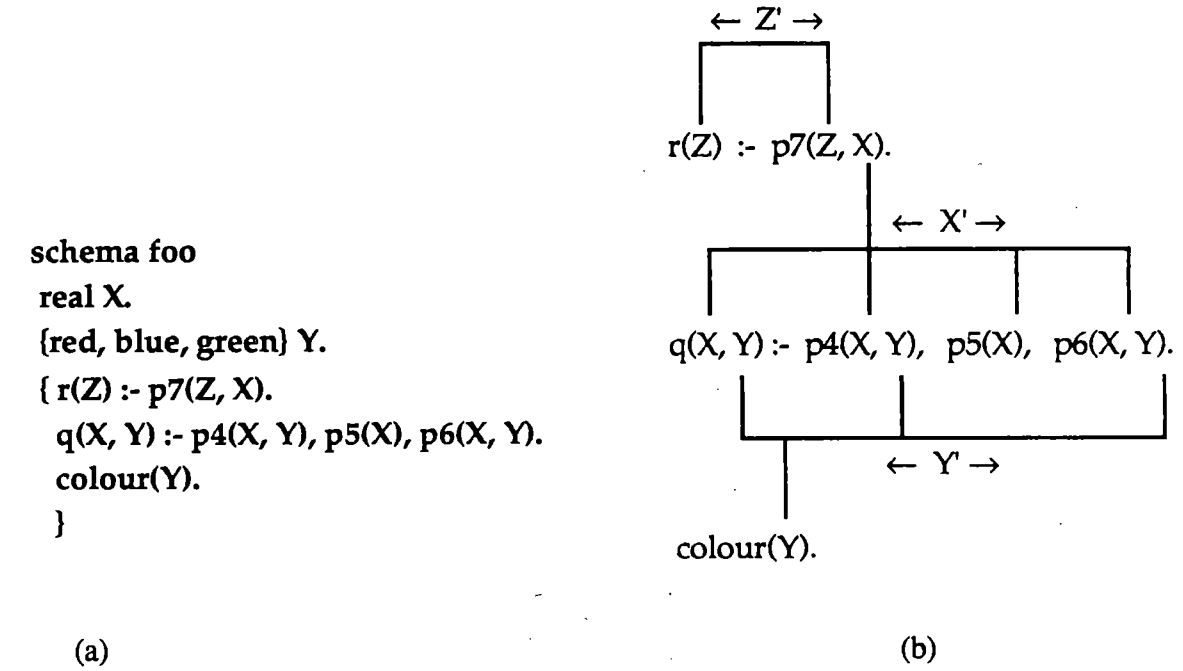


Figure 2: Dataflow over Persistent Object Variables.

The dataflow diagram for foo is drawn in Figure 2b. The persistent variables serve to link clause instances across different proof trees. Indeed the clauses do not even have to be associated with the same query. The complications for realizing dependency backtracking are serious. We are forced not only to propagate constraints across multiple proof trees but also to backtrack across this same structure. Given a particular culprit in the *proof network*, we can no longer afford to undo every subsequent goal in the chronological ordering. Standard dependency backtracking will not suffice. It would require undoing every message sent to any schema subsequent to the message which contains the culprit. For interactive reasoning tasks, this would be intolerable. We must attempt instead to analyze the dataflow of backtracking information throughout the proof network thereby retaining as much of the work as possible.

3. Hypothetical Dataflow

The difficulties in augmenting CLP with dependency backtracking noted above lead towards a notion of *hypothetical dataflow*. We visualize the CLP reasoning engine as systematically constructing

hypotheses by making nondeterministic goal choices then propagating the logical consequences of these choices over the clauses constructed to represent the proof tree(s). The nodes in a proof tree correspond to goals and constraints. For each goal, the reasoning engine makes a free choice of a program clause among the candidate set for that predicate. The chosen clause is unified with the goal. On success, the unification establishes new identity constraints between the corresponding terms in the goal and the clause. By the addition of these new constraints, hypothetical information is propagated along the logical variables on both sides, potentially in both directions. Persistent object variables allow hypothetical information to pass between proof trees established for different queries. Hypothetical information is propagated along variables but through constraints, predicates and across the identity constraints linking clauses. When it is propagated from one variable to another, it is transformed by the constraint or clause into a different form but it inherits the underlying hypotheses which created it.

In this section, we describe the nature of this hypothetical information and how it can be used in a CLP reasoning engine. In the next section, we show how it can be used to realize dependency backtracking.

3.1 Hype

The nature of the hypothetical information is easy to characterize. It is composed of two parts: 1) the current binding of an associated logical variable, and; 2) the set of goal choices which have caused this particular binding. Hypothetical information is represented as discrete units associated with logical variable events which we refer to simply as *hype* for lack of a better term. The hype associated with variable X is the quantity:

$$\eta = (X', H)$$

where X' is the refined binding of X and H is the set of hypotheses supporting the new binding. H is called the *label* of η using normal RMS terminology [7]. We know that CLP reasoner has applied some constraint to X yielding a new binding X' such that:[6]

$$X' \subset X$$

The normal mode for the CLP reasoner is to progressively refine X from its full declared domain D_{X0} towards a ground value "a" or perhaps too far to *bottom*. We can write:

$$X^0 \supset X' \supset \ldots \supset \{a\} \supset \bot.$$

Each step in this refinement requires adding new hypotheses to the label supporting the new binding. The label for bottom represents an inconsistent environment, a nogood.

[6]We assume all logical variables are domain variables of one type or another. The binding for a domain variable is some subset of its original declared domain.

3.2 Goal Metavariables

Hypothesis is created freely in the process of satisfying goals. We represent the nondeterministic state of each goal in the proof tree for a particular query as a state variable. These state variables are not variables in the underlying first-order logic (*i.e.* the object language) but are *goal metavariables*. They are accessible only to the reasoning engine itself. Each metavariable has an associated domain which contains the clause choices which may be used to satisfy the goal. Choosing a particular value for the metavariable creates a new hypothesis about the proof of the query. The choice causes the corresponding clause to be unified with the goal and consequently the proof tree to be further elaborated.

The reasoning engine maintains a vector of the goal metavariables which it manipulates to find solutions to the input queries. In the process, goals are elaborated, predicates are evaluated and constraints are propagated. The size of the state vector grows as a proof tree is constructed and shrinks when dependency backtracking occurs. Metavariables are freely manipulated by the reasoning engine to effect dependency backtracking. Indeed, the engine is only concerned with assigning metavariables such that the clauses in the proof tree remain consistent during elaboration.

Let $G = p(X_1, \ldots, X_n)$ be an unelaborated goal where p is the principle functor and $X_1, \ldots, X_n$ are the arguments. We associate with G a goal metavariable ζ which records the current clause choice. The domain of ζ is $D_\zeta = \{p^1, \ldots, p^m\}$ which is the set of clauses defining the predicate p. When G is selected for elaboration, it will be satisfied if some clause $p^i \in D_\zeta$ can unify with G such that all its subgoals can be satisfied and all its constraints remain consistent. The process of choosing p^i is nondeterministic in the general case.

The representation of D_ζ is mutable in order that it can be manipulated by the reasoning engine and its RMS. Associated with D_ζ is a vector which has the following properties. Each element j in the vector may contain either nil (indicating that $p^j \in D_\zeta$ is a valid candidate clause for G) or a *nogood* (which indicates that $p^j \notin D_\zeta$ and the clause is an inconsistent choice for G).

Elements of D_ζ are deleted by various operations in the reasoning engine. Retrieval of candidates clauses from the knowledge base deletes those elements which cannot match the goal pattern of G. The unification of G and a candidate clause deletes the candidate if the clauses cannot be unified. Elements are also deleted on failure by the RMS as described later in §4. If ever $D_\zeta = \varnothing$, then every candidate clause for G has been attempted and failed. Consequently, the predicate $p(X_1, \ldots, X_n) = \bot$ for the arguments $X_1, \ldots, X_n$ given in G and *bottom* is signaled to the RMS. Deep backtracking is then begun with G as the inconsistent node.

Goal metavariables and their domains are only constructed for nondeterministic goals (those goals which could be satisfied by unification with more than a single clause in the knowledge base). Neither are metavariables constructed for primitive constraints or primitive system predicates. Generators such as *split(X)* and *indomain(X)* defined as in CHIP [26] also have associated metavariables but of a different type.

3.3 Labels

We use the notion of label for the same purpose as [9, 1]. Labels record causal support in the RMS for dependency backtracking. A label is a set of clause choices which has caused a particular derivation. More precisely, a label is a chronologically ordered set of metavariable choices:

$$H = \{ \zeta_w=a_w, \ldots, \zeta_i=a_i, \ldots, \zeta_c=a_c \} \quad \text{for } w < i < c$$

Each metavariable choice is an assignment, $\zeta_i=a_i$, where ζ_i is the metavariable and $a_i \in D\zeta_i$ is its choice value. In any non-empty label, there is a distinguished choice, $\zeta_c=a_c$, which is the most recent called the *culprit* [25] which is the choice selected on failure for dependency backtracking. Besides being associated with logical variables, labels are also attached to boolean propositions, and first order goals:

3.3.1 Propositions

The nodes in a proof tree correspond to subgoals generated by a particular derivation (and hence a particular choice of metavariables). These choices are independent of the support for any logical variables which may appear as arguments in their goals. We define H(?p) as the label of the proposition, ?p, associated with some goal:

$$G = p(X_1, \ldots, X_n)$$

H(?p) is the ordered set of metavariables choices (from a toplevel query to the goal G itself) which have caused G to be elaborated. Each choice,

$$\zeta_Q=i \in H(?p),$$

specifies a particular choice of a clause, $q^i \in Q$, to satisfy either G or one of its parent goals. If any choice in H(?p) is changed, it will necessarily remove G from the proof tree. For a purely propositional system, H(?p) entirely characterizes the state of G. This is not true for first order goals where the support for their arguments must also be considered.

3.3.2 Goals

The label H(G) of the goal G is the set of metavariable choices which have caused the goal to be in its current state of resolution. H(G) is composed of two different influences:

- the propositional label, H(?p), for G, and
- the variable label, H(X), for each variable, $X \in \{X_1, \ldots, X_n\}$.

We can write:

$$H(G) = H(?p) \cup H(X_1) \cup \ldots \cup H(X_n)$$

where contributions of support from the various components of H(G) are not disjoint but usually share

many of the same metavariables.

Given the goal G and the candidate clauses in D_ζ, the reasoning engine must select one clause to unify with G. We think of the reasoning engine as having an active (and clever) agent associated with each goal. This agent continually monitors its metavariable. Whenever its current value becomes "nogood", the agent spontaneously changes the value. It does so by examining the remaining possible choices in the metavariable domain. It chooses a next candidate clause (by assigning a new value to the metavariable) such that the substitution of the new clause for the previous failed clause will minimize the constraint propagation effects to the existing proof tree(s).

Let $j \in D_\zeta$ be chosen as the current value for ζ. Let the corresponding rule clause p^j be of the form:

$$p(Y_1, \ldots, Y_n) :- R_1, \ldots, R_i, \ldots, R_m.$$

If the unification fails, $p^j(X_1, \ldots, X_n) = \perp$ meaning that clause p^j is false for arguments $(X_1, \ldots, X_n)$. The reasoner deletes value j from D_ζ and looks for another choice. The nogood which supports the deletion in D_ζ is derived from H(G) as described in §4.

If the unification succeeds, the reasoning engine constructs new metavariables for each goal R_i in the body of p^j. The propositional label, H(?Ri), for Ri is:

$$H(?R_i) = H(?p) \cup \{\zeta = j\}$$

3.4 Combination Rules

New hype is constructed by unification, constraint propagation and predicate application. Dataflow dependency depends on the *actual* changes made to logical variables propagated among related goals, not the potential of changes [27]. Sharing a variable between two goals is necessary but not sufficient for data dependency between them. Indeed in CLP languages, the dataflow dependency can run in both directions simultaneously between goals sharing common variables.

3.4.1 Unification

The propagation of hype is initiated by unification (which is induced by metavariable choices). Unification builds *identity constraints* between unifying terms. We consider four cases of dataflow resulting from unification which are illustrated in Figure 3 in sequence. Let X and Y both be arbitrary terms to be unified. During their unification, hype can be exchanged between X and Y as follows:

• **case 1: X == Y** - Both terms unify exactly. A new identity constraint is established between X and Y.[7] No monotonic change need occur to either term and no hype is propagated in either direction. If X and Y are logical variables, then DX = DY and the labels of both variables remain the same after unification:

$$H'(X) = H(X) \quad \text{and} \quad H'(Y) = H(Y).$$

[7]The identity constraint may be implemented by both variables sharing common storage.

• **case 2:** $\mathbf{X \Rightarrow Y}$ - The variable Y subsumes the term X causing hype to propagate from X to Y. If X is also a variable, then $D_X \subset D_Y$. The new hype $\eta_{Y'}$ is derived from X and the proposition, ?p, (which caused the unification) as follows:

$$\eta_{Y'} = (X, H(X) \cup H(?p))$$

$\eta_{Y'}$ is propagated to Y with the result that:

$$Y' = X \text{ and } H'(Y) = H(X) \cup H(?p).$$

The label of X remains unchanged:

$$H'(X) = H(X).$$

• **case 3:** $\mathbf{X \Leftarrow Y}$ - The variable X subsumes the term Y which is the symmetric case of 2 above. Hype is propagated from Y to X. If Y is a variable, then $D_X \supset D_Y$. Hype from the unification is:

$$\eta_{X'} = (Y, H(Y) \cup H(?p))$$

and is propagated to X with the result that:

$$X' = Y \text{ and } H'(X) = H(Y) \cup H(?p)$$

while the label of Y remains unchanged:

$$H'(Y) = H(Y).$$

• **case 4:** $\mathbf{X \Leftrightarrow Y}$ - Both X and Y are variables such that $D_X \cap D_Y \neq \varnothing$ and neither variable subsumes the other. New hype for both variables is the same:

$$\eta_{X'} = \eta_{Y'} = (X \cap Y, H(X) \cup H(Y) \cup H(?p))$$

which is propagated in both directions with the result that:

$$X' = Y' = X \cap Y$$

The labels for both variables are changed to:

$$H'(X) = H'(Y) = H(X) \cup H(Y) \cup H(?p).$$

Once established by unification, identity constraints are propagated like other constraints as described in the next section.

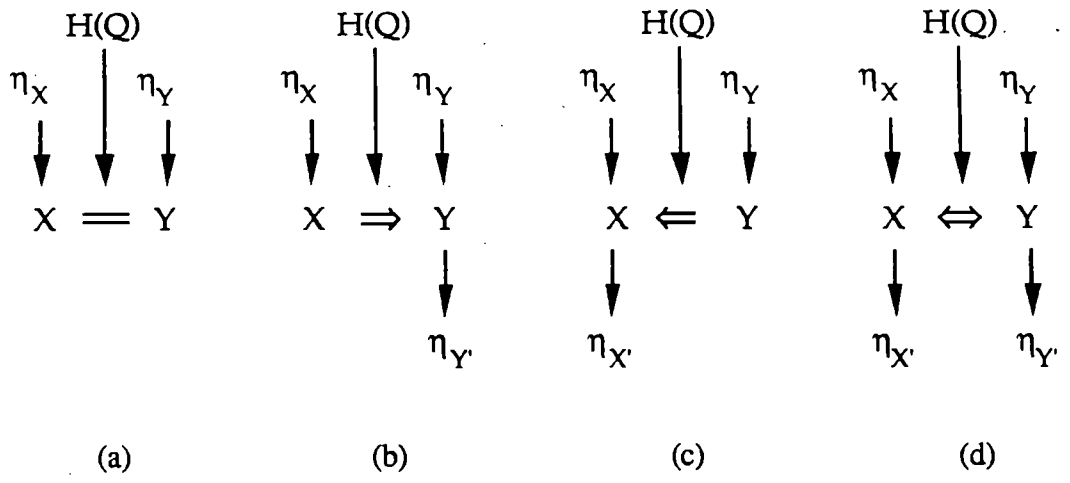


Figure 3: Dataflow under Unification

3.4.2 Constraint Propagation

Constraints tie together logical variables under a specified relation. The relation can be a primitive constraint or represented as a definite clause. Echidna uses a type of *arc consistency* [22] generalized for k-ary constraints. Constraints are implemented as compiled set functions over the variable domains. Let R be a k-ary constraint over domains D_1,..., D_k, then the i^{th} set function of R is:

$$F_i(R) = \{ a_j \mid (a_1, ..., a_j, ..., a_k) \in R, a_1 \in D_1, ..., a_i \in D_i, ..., a_k \in D_k \}$$

Graphically, the constraints are directed hyper-arcs. Each constraint has k *argument nodes* and a distinguished member of the arguments called the *target node*. The set function is applied to the argument domains yielding a new domain for the target.

Constraints transform hype across logical variables. Let C be an instance of a constraint with arguments, X_1, ... , X_i, ... , X_k:

$$C = R(X_1, ..., X_i, ..., X_k)$$

A constraint is quiescent whenever all its variables are arc consistent. X_i is made arc consistent by applying $F_i(R)$ to D_1, ... , D_i, ... , D_k thereby removing those values from D_i which do not satisfy the constraint R.

Whenever X_i is refined to X_i' by arc consistency, then new hype is added to that variable and the hype is propagated to every other constraint (either primitive or clausal) which has X_i as an argument. The new hype computed for X_i is:

$$\eta_{X'_i} = (X_i', H(C))$$

where H(C) is the label supporting the application of the constraint:

$$H(C) = H(?c) \cup H(X_1) \cup ... \cup H(X_{i-1}) \cup H(X_{i+1}) \cup ... \cup H(X_k)$$

Graphically, the propagation of hype for constraints is illustrated in Figure 4. The propositional support for the constraint H(?c) is inherited from the goal clause Q in which it appears:

$$H(?c) = H(?Q).$$

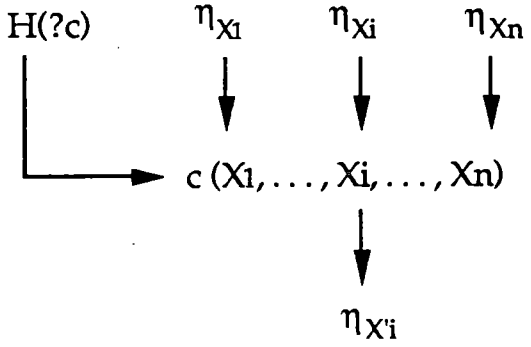


Figure 4: Dataflow through Constraints

3.4.3 Predicate Application

Predicates include both system primitive predicates and definite clauses (without embedded constraints). Predicates are elaborated once by the reasoning engine causing the proof tree to grow new branches. Hence, a predicate can only create new hype once but for more than a single argument variable. Let p be a predicate on arguments X_1, ... , X_i, ... , X_n. The predicate is applied to its arguments and for each argument X_i which is bound to a new value X_i', we create new hype:

$$\eta_{X'i} = (X_i', H(?p) \cup H(X_1) \cup H(X_i) \cup H(X_n))$$

and propagate it along X_i. The general case of dataflow for predicates is illustrated in Figure 5 where only some of the $\eta_{X'i} \neq \eta_{Xi}$.

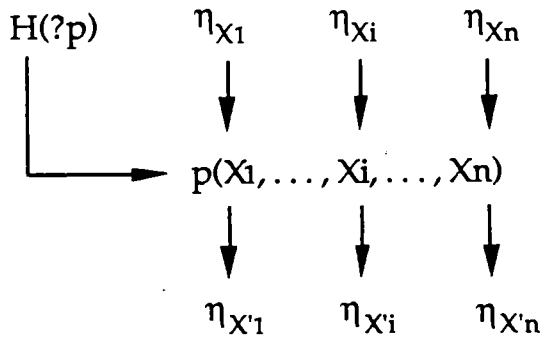


Figure 5: Dataflow in Predicates

4. Dataflow Dependency Backtracking

Given the machinery described above, intelligent backtracking is straightforward to implement. We make three significant contributions:

- The culprit is identified as the most recent clause choice which has bound a logical variable appearing in some inconsistent constraint or failed predicate. The method is analogous to unification context analysis of [27] but extended for domain variables in a CLP reasoner.
- Like Drakos[9], nogood information is computed during forward reasoning (and constraint propagation) but discarded on failure thereby limiting the size of the dependency lattice in the RMS.
- Definite clause instantiations in the proof tree are reused whenever possible thereby avoiding the recomputation of significant portions of the tree.

In our implementation, the RMS records hype events associated with logical variables. The event records form the nodes in the dependency lattice maintained by the RMS. The arcs in the lattice are the causal support of the unifications, constraint propagations and predicate applications as described above. Each node and its supporting arcs roughly correspond to a *datum* as defined by deKleer[7]. The maximal nodes of the lattice are queries and the minimal nodes are the fringe of goal elaboration and constraint propagation.

RMS uses the dependency lattice for two purposes: 1) computing the culprit on failure for dependency backtracking and; 2) optimizing the reuse of existing clauses on failure. We describe only the first function here.

The CLP reasoning system continues to elaborate goals and apply constraints until a failure or inconsistency is encountered. From the failing unification, goal or constraint, a nogood label $H(\perp)$ is constructed. $H(\perp)$ represents a metavariable choice environment which is known to be inconsistent. At least one of the choices in $H(\perp)$ must be changed to remove the failure. Any one will do since every choice was necessary to arrive at the current inconsistent proof. However, in order to ensure completeness in dependency backtrack search, we synchronize the iteration of choice variables by selecting the most recent chronological choice in $H(\perp)$ as the culprit. Let $H(\perp)$ take the form:

$$H(\perp) = \{ \ldots, \zeta_{n-1}=a_{n-1}, \zeta_n=a_n \} \quad n > 0.$$

We identify the culprit as $\zeta_n=a_n$ and the support for removing choice a_n from $D\zeta_n$ as:

$$\Psi = \{ \ldots, \zeta_{n-1}=a_{n-1} \} \quad n \geq 0.$$

Deleting a_n from $D\zeta_n$ has the side effect of undoing the clause instance allocated for that choice. Ψ removes the clause choice from future consideration until any its own support is deleted by a higher order failure in the chronological order of $H(\perp)$. If any support for Ψ is deleted, then clause a_n is restored to metavariable domain $D\zeta_n$ and becomes a candidate clause for G again. Note that current dependency backtracking techniques in Prolog reinitialize the entire goal including all of its candidate clauses in order to maintain the lexical order of clause selection defined in the language. We make no assumptions in Echidna about the execution order of candidate clauses for a goal.[8] This approach gains efficiency because it only restores those candidate clauses which may possibly now succeed given the higher failure.

Once culprit has been identified and its current metavariable choice deleted with the support of the nogood, the goal is left again unsatisfied and unelaborated. A new value is chosen arbitrarily for the metavariable from its domain (if any candidate clauses remain) and unification begun anew. On success, new identity constraints are established among the unifying terms in the goal and the matching clause and their bindings propagated.

If the culprit metavariable domain is empty, then no possible alternate clause can be unified with its goal. Every clause in the metavariable domain has been tried and failed. This initiates deep backtracking because the goal itself is false given its arguments. $H(G) = \perp$. The union of all the individual nogoods for the metavariable choices is itself a nogood [1]. The corporate environment represented by these choices necessarily leads to failure. We construct the nogood for G as follows:

$$\text{nogood}(G) = \bigcup_{p^i \in D_\zeta} \text{nogood}(p^i)$$

then identify the new culprit as the most recent choice in this ordered set. And the dependency backtracking process continues.

5. Conclusion

In this short paper, we have described a method for realizing efficient dependency backtracking in a new CLP language called Echidna. The new language is intended for model-based expert system applications which require object-centered knowledge representation, support for hypothetical reasoning

[8]The programmer must avoid assumptions about the execution order of clauses in a predicate.

and an interactive user interface. To achieve these capabilities, we have synthesized techniques from logic programming, constraint reasoning, reason maintenance and object-oriented programming. Previous dependency backtracking techniques have been applied to logic programming experimentally. Extending them to CLP has presented some difficulty which is further exacerbated by persistent object variables. These problems have necessitated development of a new way of viewing the propagation of hypothetical information in CLP reasoners. The result is dataflow dependency backtracking. A first implementation of the Echidna language is complete with a second commercial version underway. We are currently experimenting with the new programming language in a number of applications. Our theoretical interest is now focused on optimizing backtrack efficiency by reusing existing clause instances on failure even when their arguments change nonmonotonically.

Acknowledgements

The author would like to acknowledge support for this research from the Centre for Systems Science at SFU, the Science Council of British Columbia, and the Natural Sciences and Engineering Research Council of Canada.

References

[1] M. Bruynooghe & L.M. Pereira (1984) Deduction Revision by Intelligent Backtracking, in J.A. Campbell (ed.) *Implementations of Prolog*,Ellis Horwood Publishers.

[2] J.-H. Chang & A. M. Despain (1985) Semi-Intelligent Backtracking of Prolog Based on a Static Dependency Analysis, *proc. IEEE Symp. of Logic Programming*, Boston, July 1985, pp.10-21.

[3] W.Chen & D.S. Warren (1988) Objects as Intensions, *proc. 5th Int. Comp & Symp. on Logic Programming*, MIT Press, Cambridge, Mass.

[4] A. Colmerauer (1990) An Introduction to Prolog III, *Communications of the ACM 33 (7)*, pp. 69-90.

[5] R. Davis (1984) Diagnostic Reasoning Based on Structure and Behavior, *Artificial Intelligence 24 (3)*.

[6] R. Dechter (1990) Enhancement Schemes for Constraint Processing: Backjumping, Learning and Cutset Decomposition, *Artificial Intelligence 41(3)*, pp.273-312.

[7] J. deKleer (1986) An Assumption-Based TMS, *Artificial Intelligence 28(2)*,pp.127-162.

[8] J. Doyle (1979) A Truth Maintenance System, *Artificial Intelligence 12*, pp.231-272.

[9] N. Drakos (1988) Reason Maintenance in Horn Clause Logic Programs, in Smith & Kelleher (eds.) *Reason Maintenance Systems and their Applications*, John Wiley and Sons, Toronto.

[10] E. C. Freuder (1978) Synthesizing Constraint Expressions, *Communications of the ACM 21 (11)*, pp. 958-966.

[11] J. Gaschnig (1979) Performance Measurement and Analysis of Certain Search Algorithms, Tech. Rep. CMU-CS-79-124, Carnegie Mellon University, Pittsburgh, Pa.

[12] M. R. Genesereth (1984) The Use of Design Descriptions in Automated Diagnosis, *Artificial Intelligence 24 (3)*.

[13] R. M. Haralick & G. L. Elliot (1980) Increasing Tree-Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence 14*, pp.263-313.

[14] W. S. Havens (1991) Improved Dependency Backtracking for CLP Languages with Persistent Variables, (in preparation).

[15] W.S. Havens, J. Dill, J Dickinson, J. Crawford, I. Begg, R. Kheraj & W. Moore (1991) Intelligent Graphics Interfaces for Supervisory Control and other Real-Time Database Applications, proc. 7th Int. Conf. on Data Eng., 8-12 April, 1991, Kobe, Japan (to appear).

[16] W.S. Havens, J.D. Jones, C. Hunter, S. Joseph & A. Manaf (1991) Applications of the Echidna Constraint Reasoning System, in C. Y. Suen & R. Shinghal (eds.) Operational Expert Systems in Canada, Pergamon Press (in press).

[17] W. S. Havens & P. S. Rehfuss (1989) Platypus: a Constraint-Based Reasoning System, proc. 1989 Joint Int.Conf. on Artificial Intelligence, Detroit, Mich., August, 1989.

[18] W. S. Havens (1990) Echidna Constraint Reasoning System: Programming Specifications, *proc. Computational Intelligence '90,* 24-28 Sept. 1990, Milano, Italy.

[19] W. S. Havens, S. Sidebottom, M. Cuperman, R. Davison, P. MacDonald, G. Sidebottom (1990) *The Echidna Constraint Reasoning System (Version 0): Programming Manual,* CSS-IS Tech. Report 90-7, Centre for Systems Science, Simon Fraser University, Burnaby, B.C., Canada.

[20] V. Kumar & Y. Lin (1988) A Data-Dependency-Based Intelligent Backtracking Scheme for Prolog, *Journal of Logic Programming 5(2),* pp.165-181.

[21] J. Jaffar & J-L. Lassez (1987) Constraint Logic Programming, in *POPL-87,* Munich, January 1987.

[22] A.K. Mackworth (1977) Consistency in Networks of Relations, *Artificial Intelligence 8 (1),* pp.99-118.

[23] S. Mittal, C.L. Dym & M. Morjaria (1986) PRIDE: an Expert System for the Design of Paper Handling Systems, *Computer,* July, 1986, pp.102-114.

[24] B. A. Nadel (1989) Constraint Satisfaction Algorithms, *Computational Intelligence 5,* pp.188-224.

[25] R.M. Stallman & G.J. Sussman (1977) Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis, *Artificial Intelligence 9,* p.135.

[26] P. Van Hentenryck (1989) *Constraint Satisfaction in Logic Programming,* MIT Press, Cambridge, Mass.

[27] J.-H. You & B. Wong (1989) Intelligent Backtracking in Prolog Made Practical, tech. rep. TR 89-4, Dept. of Comp. Science, U. of Alberta, Edmonton, Alberta, Canada, January 1989.

[28] Waltz, D. L. (1972) Generating Semantic Descriptions from Drawings of Scenes with Shadows, Technical Report, MIT.

[29] D.H.D. Warren (1977) Implementing Prolog: Compiling Predicate Logic Programs, D.A.I. Research Report nos. 39 & 40, Univ. of Edinburgh, Scotland.

[30] C. Zaniolo (1984) Object-Oriented Programming in Prolog, proc. Int. Symp. on Logic Programming, IEEE Computer Society, pp.265-270.

Operational Semantics of Constraint Logic Programming over Finite Domains

Pascal Van Hentenryck
Brown University, Box 1910,
Providence, RI 02912
Email: pvh@cs.brown.edu

Yves Deville
University of Namur, 21 rue Grandgagnage
B-5000 Namur (Belgium)
Email: yde@info.fundp.ac.be

Abstract

Although the Constraint Logic Programming (CLP) theory [7] is an elegant generalization of the LP theory, it has some difficulties in capturing some operational aspects of actual CLP languages (e.g. [8, 6, 3, 12, 18, 19]). A difficulty comes from the intentional incompleteness of some constraint-solvers. Some constraints are simply delayed until they can be decided by the constraint-solver. Others are approximated, providing an active pruning of the search space without being actually decided by the constraint-solver.

This paper presents an extension of the *Ask & Tell* framework [14] in order to give a simple and precise operational semantics to (a class of) CLP languages with an incomplete constraint-solver. The basic idea is to identify a subset of the constraints (the basic constraints) for which there exists an efficient and complete constraint-solver. Non-basic constraints are handled through two new combinators, relaxed ask and relaxed tell, that are in fact relaxations of the standard ask and tell.

The extended framework is instantiated to CLP on finite domains, say CLP(F) [16, 6]. Arc-consistency is shown to be an efficient and complete constraint-solver for basic constraints. We also present how non-basic constraints can be approximated in CLP(F). The resulting semantics precisely describes the operational semantics of the language, enables the programmer to reason easily about the correctness and efficiency of his programs, and clarifies the links of CLP(F) with the CLP and *Ask & Tell* theories.

It is believed that the approach can be used as well to endow other CLP languages such as BNR-Prolog [12], CLP(Σ^*) [19], and parts of Trilogy [18] with a precise operational semantics.

1 Introduction

Constraint Logic Programming (CLP) is a generalization of Logic Programming (LP) where unification, the basic operation of LP languages, is replaced by the more general concept of

constraint-solving over a computation domain.

Syntactically, a CLP program can be seen as a finite set of clauses of the form

$$H \leftarrow c_1 \wedge \ldots \wedge c_n \diamond B_1 \wedge \ldots \wedge B_m$$

where $H, B_1, \ldots, B_m$ are atoms and $c_1, \ldots, c_n$ are constraints. A goal in a CLP language is simply a clause without head.

The declarative semantics of a CLP language can be defined either in terms of logical consequences or in an algebraic way. An answer to a CLP goal is no longer a substitution but rather a conjunction of constraints $c_1, \ldots, c_n$ such that

$$P, \mathcal{T} \models (\forall)(c_1 \wedge \ldots \wedge c_n \Rightarrow G) \text{ (logical version)}$$
$$P \models_{\mathcal{S}} (\forall)(c_1 \wedge \ldots \wedge c_n \Rightarrow G) \text{ (algebraic version)}$$

where P is a program, $\mathcal{S}$ is a structure, $\mathcal{T}$ is the theory axiomatizing $\mathcal{S}$, and $(\forall)(F)$ represents the universal closure of F. The rest of the presentation can be read from a logic or algebraic point of view and we use the notation $\mathcal{D} \models$ to denote that fact.

The operational semantics of CLP amounts to replacing unification by constraint-solving. It might be defined by considering configurations of the form $\langle G, \sigma \rangle$ where G is a conjunction of atoms and σ is a consistent conjunction of constraints. Now the only transition that needs to be defined between configurations is the following:

$$\frac{H \leftarrow c_1 \wedge \ldots \wedge c_n \diamond B_1 \wedge \ldots \wedge B_m \in P \qquad \mathcal{D} \models (\exists)(\sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A)}{\langle A \wedge G, \sigma \rangle \longmapsto \langle B_1 \wedge \ldots \wedge B_m \wedge G, \sigma \wedge c_1 \wedge \ldots \wedge c_n \wedge H = A \rangle}$$

In the above transition rule, A is the selected atom (it can be any atom in the goal since the order in a conjunction is irrelevant) and $(\exists)(F)$ represents the existential closure of F.

The CLP theory [7] imposes a number of restrictions on $\mathcal{S}$, $\mathcal{T}$, and their relationships to establish equivalences between the semantics. Also the CLP language should be embedded with a *complete* constraint-solver, which means that, given a conjunction of constraints σ, the constraint-solver should return *true* if $\mathcal{D} \models (\exists)(\sigma)$ and *false* otherwise (i.e $\mathcal{D} \models (\forall)(\neg\sigma)$).

Although the CLP theory is an elegant generalization of the LP theory, it has some difficulties in capturing all operational aspects of actual CLP languages such as [8, 6, 3, 12, 18, 19]. One difficulty comes from the possibility offered by these languages to state constraints that cannot be decided upon by the constraint-solver. This is the case for instance of non-linear constraints over rational and real numbers which are simply delayed until they become linear. Another difficulty comes from the fact that, for some classes of problems, an intentionally incomplete but efficient constraint-solver might well turn out to be more valuable from a programming standpoint than a complete constraint-solver. This is justified by the trade-off, that appears in many combinatorial problems, between the time spent in pruning and searching.

The present paper originates from an attempt to define in a precise and simple way the operational semantics of CLP over finite domains, say CLP(F) [16, 6], which suffers for both difficulties mentioned above. CLP(F) was designed with the goal of tackling a large class of combinatorial optimization problems with a short development time and an efficiency competitive with procedural languages. Typical applications for which CLP(F) is successful include sequencing and scheduling, graph coloring and time-table scheduling, as well as various assignments problems. The basic idea behind CLP(F) is to associate to variables a domain which is a finite set of values. Constraints in CLP(F) include (possibly non-linear) equations, inequalities, and disequations. It is simple to see that a complete constraint-solver exists for the above constraints but would necessarily require exponential time (unless P = NP). Moreover many of the abovementioned applications require different solutions and use various kinds of constraints, heuristics, and problem features. Hence it is unlikely that a complete constraint-solver be adequate for all of them. Fortunately most of them basically share the same pruning techniques and the idea behind CLP(F) was precisely to provide the language with those techniques[1]. However CLP(F) does not inherit directly its operational semantics from the CLP framework since, on the one hand, some constraints are only used when certain conditions are satisfied (e.g. disequations) and, on the other hand, some constraints are used to prune the search space although the constraint-solver cannot, in general, decide their satisfiability (e.g. inequalities). Previous approaches to define the operational semantics of CLP(F) were not based on the CLP framework but rather were given in terms of inference rules [15].

The new operational semantics presented here is based on the *Ask & Tell* framework [14] which generalizes the CLP framework by adding the concept of constraint entailment (i.e. ask) to the concept of constraint-solving (i.e. tell). Ask constraints directly account for the first difficulty in CLP languages: they might be used to restrict the context in which a constraint is executed. However to account for the incompleteness of the constraint-solver, we need to generalize the framework[2]. The basic idea behind the semantics presented here is to split the set of primitive constraints into two sets:

- *basic* constraints for which there exists an efficient and complete constraint-solver;
- *non-basic* constraints which are only approximated.

The basic constraints are handled following the standard CLP theory and are precisely those constraints that can be returned as an answer to a goal. Non-basic constraints are handled through two new combinators, relaxed ask and relaxed tell. Contrary to ask (resp. tell), relaxed ask (resp. relaxed tell) check entailment (resp. consistency) not wrt the accumulated constraints but rather wrt a relaxation of them. Moreover relaxed tell enables to deduce new basic constraints approximating the non-basic one. Formally, relaxed ask

[1]Note that for other types of problems a complete constraint-solver might be more appropriate [1].

[2]In fact Saraswat considers his framework as parametrized on the combinators as well so that we are actually instantiating his framework.

and relaxed tell are not mere combinators but rather they define a family of combinators parametrized by a relaxation and an approximation function. The relaxation function specifies the relaxation of the accumulated constraints while the approximation specifies how to infer new basic constraints.

Describing a CLP language in the extended framework (i.e. with relaxed ask and tell) amounts to specifying:

- the basic constraints and their associated complete constraint-solver;
- the non-basic constraints and their associated (1) relaxation functions, (2) approximation functions, and (3) constraint-solvers that are complete for the conjunction of a non-basic constraint and the relaxation of a conjunction of basic constraints.

The extended framework is then instantiated to CLP(F). We identify the subset of basic constraints and show that arc-consistency [9] provides an efficient and complete constraint-solver for them. We also define the relaxation and approximation functions used in CLP(F) and suggest the non-basic constraint-solvers.

The contributions of the paper are twofold.

1. It gives a precise and simple operational semantics to CLP(F), characterizing what is computed (e.g. what is an answer) and how the computation is achieved (e.g. what is the pruning at some computation point). The semantics allows the programmer to formally reason about the correctness and the efficiency of his programs. It also clarifies the relationships between CLP(F) on the one hand, and the CLP and *Ask&Tell* frameworks on the other hand.

2. It proposes an extended framework that can possibly be instantiated to endow languages such as BNR-Prolog [12], CLP(Σ^*) [19], and parts of Trilogy [18] with a precise operational semantics.

The rest of this paper is organized in the following way. The next section defines the operational semantics of basic CLP. Section 3 instantiates basic CLP to finite domains. Section 4 describes the new combinators for non-basic constraints, relaxed ask and relaxed tell. Section 5 instantiates relaxed ask and relaxed tell to CLP(F). Section 6 contains the conclusion of this paper and directions for future research.

2 Basic CLP

In this section, we define the syntax and operational semantics of the class of languages we consider for Constraint Logic Programming. We use a structural operational semantics [13] similar to the one used in [14]. There is an important difference however due to the various possible interpretations of nondeterminism. Saraswat's semantics, as well as other semantics for concurrent logic programming languages, describes all possible executions of a program

on a given goal. Hence an actual implementation might actually fail (resp. succeed) for some elements of the success (resp. failure) set. Our semantics captures a single execution and hence an element of the success set might never fail in an actual implementation. This difference does not show up in the transition rules but rather in the definition of the success, failure, flounder, and divergence sets.

2.1 Syntax

The abstract syntax of the basic language can be defined by the following (incomplete) grammar:

$$P ::= H \leftarrow B \mid P.P$$
$$B ::= A \mid ask(c) \rightarrow B \mid tell(c) \mid B \,\&\, B \mid \exists x\, B \mid true$$

In words, a program is a non-empty set of clauses. A clause is composed of a head and a body. The head is an atom whose arguments are all distinct variables. The body is either an atom, an implication $ask(c) \rightarrow B$ where $ask(c)$ is an ask on constraint c, a tell on constraint c, a conjunction of two bodies, an existential construction $\exists x\, B$ where B is a body with variable x free, or *true*. For completeness, some semantic rules should also be added, for instance the rule stating that any variable in a clause is either existentially quantified or appears in the head.

The concrete syntax can be the one of any existing CLP language suitably enhanced to include the implication construct. There is no difficulty in translating any of these concrete syntax to the abstract one.

2.2 Basic Constraints

Basic constraints are split into two sets: basic tell-constraints, simply referred to as basic constraints, and basic ask-constraints. The constraint-solver for basic CLP should be complete for consistency of basic constraints and entailment of basic ask-constraints.

Definition 1 A basic constraint-solver is *complete* iff it can decide

1. consistency of c and σ (i.e. $\mathcal{D} \models (\exists)(\sigma \wedge c)$);
2. entailment of c' wrt σ (i.e. $\mathcal{D} \models (\forall)(\sigma \Rightarrow c')$);

where c is a basic constraint, c' a basic ask-constraint, and σ a conjunction of basic constraints.

CLP languages usually maintain constraints in a reduced form to obtain an incremental behaviour necessary to achieve efficiency[3].

[3]Most operational semantics do not include this function as it plays no fundamental role in their description. As we will see, it plays an important role in showing how efficiently the relaxation function can be computed in CLP(F).

Definition 2 A *reduction* function is a total function *red* which, given a consistent conjunction of basic constraints σ, returns a conjunction of basic constraints, such that $\mathcal{D} \models \sigma \Leftrightarrow red(\sigma)$.

In the following, we assume the existence of a reduction function *red* for basic constraints and denote by CCR the set of consistent conjunctions of constraints in reduced form (i.e. the codomain of *red*). The actual choice of the reduction function depends upon the computation domain.

2.3 Structural Operational Semantics

2.3.1 Configurations

The configurations in the transition system are of the form $\langle B, \sigma \rangle$ where B is a body and σ a conjunction of basic constraints in reduced form. Informally B represents what remains to be executed while σ represents the constraints accumulated so far. Successful computations end up with a conjunction of basic constraints in reduced form. Hence $CCR \subseteq T$. Computations may also flounder when an ask on constraint c cannot be decided upon (i.e. neither c nor $\neg c$ is entailed by the accumulated constraints). We use the terminal *flounder* to capture that behaviour.
Terminal configurations are thus described by

$$T = \{\sigma \mid \sigma \in CCR\} \cup \{flounder\}.$$

Configurations are described by

$$\Gamma = \{\langle B, \sigma \rangle \mid B \text{ is a body and } \sigma \in CCR\} \cup T.$$

A transition $\gamma \longmapsto \gamma'$ can be read as "configuration γ nondeterministically reduces to γ'".

2.3.2 Transition Rules

Goals are assumed to be resolved against clauses from some program and constraints are assumed to be checked for consistency and entailment in a given structure or theory $\mathcal{D}$. Since the structure (or theory) never changes, we simply omit it and assume that it is clear from the context. As the program changes, we make use of an indexed transition system and use $P \vdash \gamma \longmapsto \gamma'$ to denote that the transition $\gamma \longmapsto \gamma'$ takes place in the context of program P. When the transition does not depend on program P, we simply drop the prefix $P \vdash$. In the following, σ denotes a conjunction of basic constraints in reduced form, γ a configuration, B and G bodies, P a program, and c a basic constraint. These are possibly subscripted.

True: The atom *true* simply leads to a terminal configuration.

$$\langle true\ ,\ \sigma \rangle \longmapsto \sigma$$

Tell: The tell operation on a basic constraint is successful if the constraint is consistent with the accumulated constraints.

$$\frac{\mathcal{D} \models (\exists)(\sigma \wedge c)}{\langle tell(c)\,,\ \sigma\rangle \longmapsto red(\sigma \wedge c)}$$

Implication: An implication $ask(c) \rightarrow B$ never fails. If the basic ask-constraint c is entailed by the accumulated constraints, it reduces to the body B. If $\neg c$ is entailed by the accumulated constraints, the implication terminates successfully. Otherwise, the implication flounders.

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow c)}{\langle ask(c) \rightarrow G\,,\ \sigma\rangle \longmapsto \langle G, \sigma\rangle}$$

$$\frac{\mathcal{D} \models (\forall)(\sigma \Rightarrow \neg c)}{\langle ask(c) \rightarrow G\,,\ \sigma\rangle \longmapsto \sigma}$$

$$\frac{\begin{array}{l}\mathcal{D} \models \neg(\forall)(\sigma \Rightarrow c)\\ \mathcal{D} \models \neg(\forall)(\sigma \Rightarrow \neg c)\end{array}}{\langle ask(c) \rightarrow G\,,\ \sigma\rangle \longmapsto flounder}$$

Existential Quantification: An existential quantification can be removed by replacing the quantified variable by a brand new variable.

$$\frac{\begin{array}{l}y \text{ is a brand new variable}\\ \langle G[x/y]\,,\ \sigma\rangle \longmapsto \gamma\end{array}}{\langle \exists x\, G\,,\ \sigma\rangle \longmapsto \gamma}$$

The fact that variable y be brand new can be formalized in various ways if necessary.

Conjunction: The semantics of conjunction is given here by the interleaving rule which is appropriate for CLP languages. If any of the goals in a conjunction can make a transition, the whole conjunction can make a transition as well and the accumulated constraints are updated accordingly. The conjunction flounders when both conjuncts flounder.

$$\frac{\langle G_1\,,\ \sigma\rangle \longmapsto \langle G_1'\,,\ \sigma'\rangle}{\begin{array}{l}\langle G_1 \,\&\, G_2\,,\ \sigma\rangle \longmapsto \langle G_1' \,\&\, G_2\,,\ \sigma'\rangle\\ \langle G_2 \,\&\, G_1\,,\ \sigma\rangle \longmapsto \langle G_2 \,\&\, G_1'\,,\ \sigma'\rangle\end{array}}$$

$$\frac{\langle G_1\,,\ \sigma\rangle \longmapsto \sigma'}{\begin{array}{l}\langle G_1 \,\&\, G_2\,,\ \sigma\rangle \longmapsto \langle G_2\,,\ \sigma'\rangle\\ \langle G_2 \,\&\, G_1\,,\ \sigma\rangle \longmapsto \langle G_2\,,\ \sigma'\rangle\end{array}}$$

$$\frac{\langle G_1\ ,\ \sigma\rangle \longmapsto flounder \quad \langle G_2\ ,\ \sigma\rangle \longmapsto flounder}{\langle G_1 \,\&\, G_2\ ,\ \sigma\rangle \longmapsto flounder}$$

Procedure Call with one Clause: We now consider the solving of an atom $p(t_1, \ldots, t_n)$ wrt a clause $p(x_1, \ldots, x_n) \leftarrow B$. If $B[x_1/t_1, \ldots, x_n/t_n]$ can make a transition in the context of the accumulated constraints, then so can $p(t_1, \ldots, t_n)$ in the context of the clause and the constraints.

$$\frac{p(x_1, \ldots, x_n) \leftarrow B \ \vdash\ \langle B[x_1/t_1, \ldots, x_n/t_n]\ ,\ \sigma\rangle \longmapsto \gamma}{p(x_1, \ldots, x_n) \leftarrow B \ \vdash\ \langle p(t_1, \ldots, t_n)\ ,\ \sigma\rangle \longmapsto \gamma}$$

Procedure Call with several Clauses: If an atom can make a transition in program P_1, it can obviously make a transition in the program made up of P_1 and program P_2.

$$\frac{P_1 \ \vdash\ \gamma \longmapsto \gamma'}{\begin{array}{l} P_1.P_2 \ \vdash\ \gamma \longmapsto \gamma' \\ P_2.P_1 \ \vdash\ \gamma \longmapsto \gamma' \end{array}}$$

2.4 Operational Semantics

We now define the operational semantics of the language in terms of its success, floundering, divergence, and failure sets. Let $\stackrel{*}{\longmapsto}$ denote the reflexive and transitive closure of $\longmapsto$ and $initial(G, \sigma)$ the configuration $\langle G, red(\sigma)\rangle$. Also a configuration γ is said to diverge in program P if there exists an infinite sequence of transitions

$$P \vdash \gamma \longmapsto \gamma_1 \longmapsto \ldots \longmapsto \gamma_i \longmapsto \ldots$$

The success, floundering, and divergence sets (not necessarily disjoint) can be defined in the following way.

$$\begin{array}{lcl} SS[P] & = & \{\langle G, \sigma\rangle \mid P \vdash initial(G, \sigma) \stackrel{*}{\longmapsto} \sigma'\} \\ FLS[P] & = & \{\langle G, \sigma\rangle \mid P \vdash initial(G, \sigma) \stackrel{*}{\longmapsto} flounder\} \\ DS[P] & = & \{\langle G, \sigma\rangle \mid initial(G, \sigma) \text{ diverges in P}\} \end{array}$$

The failure set can now be defined in terms of the above three sets.

$$FS[P] \ = \ \{\langle G, \sigma\rangle \mid \langle G, \sigma\rangle \notin SS[P] \cup FLS[P] \cup DS[P]\}$$

Another semantic definition can be given to capture the results of the computation.

$$RES[P]\langle G, \sigma\rangle \ = \ \{\sigma' \mid P \vdash initial(G, \sigma) \stackrel{*}{\longmapsto} \sigma'\}$$

3 Basic CLP(F)

In this section, we instantiate basic CLP to finite domains.

3.1 Basic Constraints

In the following, x and y, possibly subscripted, denote variables and a, b, c, v, w, possibly subscripted, denote natural numbers.

Definition 3 The *basic* constraints of CLP(F) are either *domain* constraints or *arithmetic* constraints.

- domain constraint: $x \in \{v_1, \ldots, v_n\}$;
- arithmetic constraints:
  - $ax \neq b$;
  - $ax = b$;
  - $ax = by + c$ $(a \neq 0 \neq b)$;
  - $ax \geq by + c$;
  - $ax \leq by + c$;

The semantics of addition, multiplication, $=$, $\leq$, $\geq$, and $\neq$ is the usual one. Clearly, the negation of each basic constraint can be expressed as a conjunction or disjunction of basic constraints. Hence all the basic constraints can also be basic ask-constraints[4]. Note that the variables appearing in arithmetic constraints are expected to appear in some domain constraints. Every variable thus has a domain. This can be seen as an implicit ask-constraint and justifies the following definition.

Definition 4 A *system of constraints* S is a pair $\langle AC, DC \rangle$ where AC is a set of arithmetic constraints and DC is a set of domain constraints such that any variable occurring in an arithmetic constraint also occurs in some domain constraint of S.

Definition 5 Let $S = \langle AC, DC \rangle$ be a system of constraints. The set D_x is the *domain* of x in S (or in DC) iff the domain constraints of x in DC are $x \in D_1, \ldots, x \in D_k$ and D_x is the intersection of the D_i's.

It follows that, provided that each variable has a domain, a conjunction of basic constraints can be represented by a system of constraints and vice versa.

We conclude this subsection by a number of conventions. If c is an arithmetic constraint with only one variable x, we say that c is *unary* and denote it as $c(x)$. Similarly, if c is an arithmetic constraint with two variables x and y, we say that c is *binary* and denote it $c(x, y)$. As usual, $c(x/v)$ and $c(x/v, y/w)$ denote the Boolean value obtained from $c(x)$ and $c(x, y)$ by replacing x and y by the values v and w respectively.

[4]Note that we may want to consider some of them as non-basic constraints for efficiency reasons.

3.2 Constraint-Solving

The constraint-solver of CLP(F), and hence of basic CLP(F), is based on consistency techniques, a paradigm emerging from AI research [9]. We start by defining a number of notions.

Definition 6 Let $c(x)$ be a unary constraint and D_x be the domain of x. Constraint $c(x)$ is said to be *node-consistent wrt* D_x if $c(x/v)$ holds for each value $v \in D_x$.

Definition 7 Let $c(x,y)$ be a binary constraint and D_x, D_y be the domains of x, y. Constraint $c(x,y)$ is said to be *arc-consistent wrt* D_x, D_y if the following conditions hold:

1. $\forall v \in D_x \; \exists w \in D_y \; c(x/v, y/w)$ holds;
2. $\forall w \in D_y \; \exists v \in D_x \; c(x/v, y/w)$ holds;

We are in position to define a solved form for the constraints.

Definition 8 Let S be a system of constraints. S is in *solved form* iff any unary constraint $c(x)$ in S is node-consistent wrt the domain of x in S, and any binary constraint $c(x,y)$ in S is arc-consistent wrt the domains of x, y in S.

We now study a number of properties of systems of constraints in solved form.

Property 9 Let $c(x,y)$ be the binary constraint $ax \geq by + c$, arc-consistent wrt $D_x = \{v_1, \ldots, v_n\}, D_y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have

1. $c(v_1, w_1)$ and $c(v_n, w_m)$ hold;
2. if $c(v_i, w_j)$ holds, then $c(v_{i+k}, w_{j-l})$ holds $(0 \leq k \leq n-i, \; 0 \leq l < j)$.

Proof

- (2): Since $v_{i+k} \geq v_i$, $w_j \geq w_{j-l}$ and a, b are natural numbers, we have that $av_{i+k} \geq av_i \geq bw_j + c \geq bw_{j-l} + c$. Hence $c(v_{i+k}, w_{j-l})$ holds.
- (1): By arc-consistency, $c(v_1, w)$ and $c(v, w_n)$ holds for some $w \in D_y$ and some $v \in D_x$.Hence, by (1), $c(v_1, w_1)$ and $c(v_n, w_m)$ hold.

□

Property 10 Let $c(x,y)$ be the binary constraint $ax \leq by + c$, arc-consistent wrt $D_x = \{v_1, \ldots, v_n\}, D_y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have

1. $c(v_1, w_1)$ and $c(v_n, w_m)$ hold;

2. if $c(v_i, w_j)$ holds, then $c(v_{i-k}, w_{j+l})$ holds $(0 \leq k < i,\ 0 \leq l \leq m - j)$.

Property 11 Let $c(x, y)$ be the binary constraint $ax = by + c$, arc-consistent wrt $D_x = \{v_1, \ldots, v_n\}, D_y = \{w_1, \ldots, w_m\}$. Assume also that $v_1 < \ldots < v_n$ and $w_1 < \ldots < w_m$. Then we have $n = m$ and $c(v_i, w_i)$ holds $(1 \leq i \leq n)$.

Proof The proof is by induction on i. Applying Properties 1 and 2 to the inequalities associated with $c(x, y)$ implies that $c(v_1, w_1)$ holds. Assume now that $c(v_i, w_i)$ holds for some $1 \leq i \leq n$. For all $v \in \{v_{i+1}, \ldots, v_n\}$, because a, b are non-zero natural numbers, we have

$$av \ > \ av_i \ = \ bw_i + c \ \geq \ bw_{i-k} + c \ (0 \leq k < i).$$

Hence $c(v, w_{i-k})$ does not hold. Similarly, for all $w \in \{w_{i+1}, \ldots, w_m\}$, $c(v_{i-k}, w)$ does not hold $(0 \leq k < i)$. The constraint $c(x, y)$ is thus still arc-consistent wrt the domains $\{v_{i+1}, \ldots, v_n\}$ and $\{w_{i+1}, \ldots, w_m\}$. Using Properties (1) and (2) again, we get that $c(v_{i+1}, w_{i+1})$ holds.

Assume now that $n < m$. Since $c(v_n, w_n)$ holds, $c(v, w_{n+1})$ does not hold for $v \in D_x$. The constraint $c(x, y)$ is thus not arc-consistent which is a contradiction. Hence $n = m$. □

The satisfiability of a system of constraints in solved form can be tested in a straightforward way.

Theorem 12 Let $S = \langle AC, DC \rangle$ be a system of constraints in solved form. S is satisfiable iff $\langle \emptyset, DC \rangle$ is satisfiable.

Proof It is clear that $\langle \emptyset, DC \rangle$ is not satisfiable iff the domain of some variable is empty in DC. If the domain of some variable is empty in DC, then S is not satisfiable. Otherwise, it is possible to construct a solution to S. By properties (1), (2), and (3), all binary constraints of S hold if we assign to each variable the smallest value in its domain. Moreover, because of node-consistency, the unary constraints also hold for such an assignment. □

It is worth noting that, in a system of constraints in solved form, all the values within the domain of a variable do not necessarily belong to a solution. For instance, in the following system

$$\langle \{x \leq y, 3x \geq 2y + 1\}, \{x \in \{2, 4, 6\}, y \in \{2, 3, 6\}\} \rangle$$

there is no solution with the value 4 assigned to x although the system is in solved form.

It remains to show how to transform a system of constraints into an equivalent one in solved form. This is precisely the purpose of the node- and arc- consistency algorithms [9].

Algorithm 13 To transform the system of constraints S into a system in solved form S':

1. apply a node-consistency algorithm to the unary constraints of $S = \langle AC, DC \rangle$ to obtain $\langle AC, DC' \rangle$;
2. apply an arc-consistency algorithm to the binary constraints of $\langle AC, DC' \rangle$ to obtain $S' = \langle AC, DC'' \rangle$.

Theorem 14 Let S be a system of constraints. Algorithm 1 produces a system of constraints in solved form equivalent to S.

We now give a complete constraint-solver for the basic constraints. Given a system of constraints S, Algorithm 15 returns *true* if S is satisfiable and $false$ otherwise.

Algorithm 15 To check the satisfiability of a system of constraints S:

1. apply Algorithm 13 to S to obtain $S' = \langle AC, DC \rangle$;
2. if the domain of some variable is empty in DC', return $false$; otherwise return *true*.

The complexity of Algorithms 13 and 15 is the complexity of arc-consistency algorithms. In [10], an arc-consistency algorithm is proposed whose complexity is $O(cd^2)$ where c is the number of binary constraints and d is the size of the largest domain. Given the form of the basic constraints, it is possible to design a specific arc-consistency algorithm whose complexity is $O(cd)$ [4] showing that basic constraints can be solved efficiently.

3.3 Reduced Form of the Basic Constraints

This subsection defines the reduced form of basic constraints in CLP(F).

Definition 16 Let σ be a consistent conjunction of basic constraints. The reduced form of σ, denoted $red(\sigma)$, is obtained as follows:

1. Let S be the system of constraints associated with σ and $S' = \langle AC', DC' \rangle$ be its solved form.
2. Let DC'' be equivalent to DC' be with only one domain constraint per variable.
3. Let AC'' be AC' without
   - the unary constraints;
   - the binary constraints $c(x, y)$ satisfying

     $$\forall v \in D_x \; \forall w \in D_y \; c(x/v, y/w)$$

     where D_x, D_y are the domains of x, y in S'.
4. $red(\sigma)$ is the conjunction of basic constraints in $\langle AC'', DC'' \rangle$.

The reduced form for the basic constraints is equivalent to the solved form since the unary constraints are node consistent (and hence implied by the domain constraints) and the binary constraints satisfying the given condition are also implied by the domain constraints.

3.4 Expressive Power

The basic constraints of CLP(F) have been chosen carefully in order to avoid an NP-Complete constraint-solving problem. For instance, allowing disequations with two variables leads to an NP-Complete problem (graph-coloring could then be expressed in a straightforward manner). Allowing equations and inequalities with three variables also leads to NP-complete problems. Finally, it should be noted that Algorithm 15 is not powerful enough to decide systems of constraints including constraints of the form $ax + by = c$.

The reason is that arc-consistency algorithms handle constraints locally while more global reasoning is necessary to check satisfiability of this class of constraints. As an example, the system

$$\langle \{x_1 + x_2 = 1, x_2 + x_3 = 1, x_3 + x_1 = 1\} \ , \ \{x_1 \in \{0,1\}, x_2 \in \{0,1\}, x_3 \in \{0,1\}\} \rangle$$

is not satisfiable although it is arc-consistent. To verify the unsatisfiability, just add the three constraints to obtain

$$2x_1 + 2x_2 + 2x_3 = 3.$$

The left member is even while the right member is odd.

4 Non-Basic CLP

The purpose of this section is to provide a systematic way to describe the operational semantics of non-basic constraints that are approximated in terms of basic constraints. As mentioned previously, approximation is present in several CLP languages.

To capture that behaviour, we introduce two new combinators in the *Ask & Tell* framework, relaxed tell and relaxed ask. These combinators are parametrized by a relaxation and an approximation function such that we are in fact defining a family of combinators.

Before starting the more formal presentation, let us give an informal account to the approach. Let σ be the accumulated constraints and assume that we face a relaxed tell on a non-basic constraint c. Relaxed tell, instead of checking the consistency of $\sigma \wedge c$ which might be quite complex in general, only checks the consistency of a relaxation of $\sigma \wedge c$. Clearly if the relaxed problem is not satisfiable, the initial problem is not satisfiable either. Moreover the solutions of the relaxed problem cannot necessarily be expressed as a conjunction of basic constraints. Hence only an approximation of the solutions, in the form of a conjunction of basic constraints, can be added to the accumulated constraints. When the approximation is not equivalent to the initial problem, the non-basic constraint cannot be removed from the goal part of the configuration. Finally, if we face a relaxed ask, then entailment is not checked wrt σ, but rather wrt to its relaxation. As a consequence, relaxed ask could return undecided (i.e. flounder) while an ask (if implemented) would have returned *true* or *false*.

4.1 Relaxation and Approximation

Relaxed tell and relaxed ask require to associate, with each non-basic constraint, (1) a relaxation function; (2) an approximation function and (3) a complete constraint-solver.

Definition 17 A *relaxation* function is a total function $r : CCR \rightarrow CCR$ such that

$$\mathcal{D} \models (\forall)(\sigma \Rightarrow r(\sigma)).$$

In other words, a relaxation function associates with each conjunction σ of basic constraints another conjunction of basic constraints that is implied by σ. Hence the relaxation of σ captures the solutions of σ and possibly some non-solutions.

We also would like to infer new basic constraints from a non-basic constraint. This is achieved through an approximation function.

Definition 18 Given a relaxation function r, an *approximation* function is a total function ap, which, given $\sigma \in CCR$ and a non-basic constraint c, returns a conjunction of basic constraints, such that $\mathcal{D} \models (\forall)((r(\sigma) \wedge c) \Rightarrow ap(\sigma, c))$.

This definition specifies that the approximation captures all solutions of $r(\sigma) \wedge c$ and possibly some non-solutions. In the following, we assume that a relaxation function and an approximation function have been defined for each constraint and denote them by the (overloaded) symbols r and ap respectively.

The non-basic constraint-solver has to satisfy a number of requirements.

Definition 19 A non-basic constraint-solver is *complete* iff it can decide

1. the consistency of c and $r(\sigma)$;
2. the entailment of c' by $r(\sigma)$

where c is a non-basic constraint, c' is a non-basic ask-constraint, and σ is a conjunction of basic constraints.

The following properties are straightforward but help understanding the transition rules.

Property 20 [Properties of relaxation and approximation]

1. $\mathcal{D} \models (\forall)((\sigma \wedge c) \Rightarrow (\sigma \wedge ap(\sigma, c))$;
2. $\mathcal{D} \models \neg(\exists)(c \wedge r(\sigma))$ implies $\mathcal{D} \models \neg(\exists)(c \wedge \sigma)$;
3. $\mathcal{D} \models \neg(\exists)(ap(\sigma, c) \wedge \sigma)$ implies $\mathcal{D} \models \neg(\exists)(c \wedge \sigma)$;
4. $\mathcal{D} \models (\forall)(ap(\sigma, c) \Rightarrow c \wedge r(\sigma))$ implies $\mathcal{D} \models (\forall)((\sigma \wedge c) \Leftrightarrow (\sigma \wedge ap(\sigma, c)))$;
5. $\mathcal{D} \models (\exists)(ap(\sigma, c) \wedge \sigma)$ and $\mathcal{D} \models (\forall)(ap(\sigma, c) \Rightarrow c \wedge r(\sigma))$ implies $\mathcal{D} \models (\exists)(c \wedge \sigma)$;
6. $\mathcal{D} \models (\forall)(r(\sigma) \Rightarrow c)$ implies $\mathcal{D} \models (\forall)(\sigma \Rightarrow c)$.

4.2 Relaxed Tell

We are now in position to define the operational semantics of relaxed tell.

Termination: Termination of relaxed tell occurs when the approximation $ap(\sigma, c)$ is equivalent to $r(\sigma) \wedge c$ which means that $r(\sigma) \wedge c$ can be represented as a conjunction of basic constraints. Moreover if $ap(\sigma, c)$ is consistent with σ, we obtain a terminal configuration.

$$\frac{\mathcal{D} \models (\exists)(\sigma \wedge ap(\sigma, c)) \quad \mathcal{D} \models (\forall)(ap(\sigma, c) \Rightarrow (r(\sigma) \wedge c))}{\langle relax\text{-}tell(c), \sigma \rangle \longmapsto red(\sigma \wedge ap(\sigma, c))}$$

Property 20.4 and 20.5 ensure respectively the equivalence of $\sigma \wedge ap(\sigma, c)$ and $\sigma \wedge c$ and the consistency of c and σ.

Pruning: Property 20.1 allows for the addition of new basic constraints (and hence pruning of the search space) provided that $r(\sigma) \wedge c$ and $ap(\sigma, c) \wedge \sigma$ be consistent, and $ap(\sigma, c)$ be not entailed by σ.

$$\frac{\mathcal{D} \models (\exists)(r(\sigma) \wedge c) \quad \mathcal{D} \models (\exists)(\sigma \wedge ap(\sigma, c)) \quad \mathcal{D} \models \neg(\forall)(ap(\sigma, c) \Rightarrow (r(\sigma) \wedge c)) \quad \mathcal{D} \models \neg(\forall)(\sigma \Rightarrow ap(\sigma, c))}{\langle relax\text{-}tell(c), \sigma \rangle \longmapsto \langle relax\text{-}tell(c), red(\sigma \wedge ap(\sigma, c)) \rangle}$$

The third condition (non-termination) imposes to keep $relax\text{-}tell(c)$ in the resulting configuration while the last condition (non-redundancy) avoids the infinite application of the rule. Progress is achieved here together with the conjunction rules because the search space is pruned by the new constraints.

Floundering: Floundering occurs when there is no termination and when the approximation is entailed by the accumulated constraints.

$$\frac{\mathcal{D} \models (\exists)(r(\sigma) \wedge c) \quad \mathcal{D} \models \neg(\forall)(ap(\sigma, c) \Rightarrow (r(\sigma) \wedge c)) \quad \mathcal{D} \models (\forall)(\sigma \Rightarrow ap(\sigma, c))}{\langle relax\text{-}tell(c), \sigma \rangle \longmapsto flounder}$$

4.3 Relaxed Ask

Relaxed ask can be defined in a straightforward way by checking entailment of the non-basic ask-constraint by the relaxation of the accumulated constraints.

$$\frac{\mathcal{D} \models (\forall)(r(\sigma) \Rightarrow c)}{\langle relax\text{-}ask(c) \rightarrow G, \sigma\rangle \longmapsto \langle G, \sigma\rangle}$$

$$\frac{\mathcal{D} \models (\forall)(r(\sigma) \Rightarrow \neg c)}{\langle relax\text{-}ask(c) \rightarrow G, \sigma\rangle \longmapsto \sigma}$$

$$\frac{\mathcal{D} \models \neg(\forall)(r(\sigma) \Rightarrow c) \quad \mathcal{D} \models \neg(\forall)(r(\sigma) \Rightarrow \neg c)}{\langle relax\text{-}ask(c) \rightarrow G, \sigma\rangle \longmapsto flounder}$$

4.4 Relations with Ask and Tell

The soundness of the transition system can be proven by structural induction on the configurations. More interesting perhaps is the relationships between ask and tell and their relaxed versions in the case where *ask* and *tell* can be decided in the computation domain. Assume that P^* is the program P where all occurrences of relaxed ask and relaxed tell have been replaced by ask and tell. We state the following property without proof.

Property 21

- $SS(P) \subseteq SS(P^*)$
- $FS(P) \subseteq FS(P^*)$

5 Non-Basic CLP(F)

In Section 2, we have presented the basic constraints of CLP(F). These constraints can be handled by an efficient complete constraint-solver but are certainly not expressive enough to be the only constraints available in CLP(F). Moreover we have shown that apparently simple extensions to the basic constraints can lead to an NP-Complete constraint-solving problem. To complete the definition of CLP(F), it remains to specify the non-basic constraints and to define (1) the relaxations, (2) the approximations, and (3) the constraint-solvers.

Definition 22 Let σ be a conjunction of basic constraints in reduced form and $\langle AC, DC\rangle$ its associated system of constraints. The relaxation $r(\sigma)$ is the conjunction

$$x_1 \in D_1 \wedge \ldots \wedge x_n \in D_n$$

such that $DC = \{x_1 \in D_1, \ldots, x_n \in D_n\}$.

In other words, the relaxation in CLP(F) simply ignores all constraints but the domain constraints. Note also that computing the relaxation does not induce any cost since the accumulated constraints are already in reduced form for incrementality purpose.

There are two approximation functions depending if full k-consistency is achieved (i.e. ap_1) or if the reasoning is only performed on the minimum and maximum values in the domains (i.e. ap_2). For some symbolic constraints, both approximations might be useful depending upon the problem at hand.

Definition 23 Let σ be a conjunction of basic constraints and c be a non-basic constraint with variables $x_1, \ldots, x_n$ such that $\mathcal{D} \models (\exists)(r(\sigma) \wedge c)$. Let $dom(r(\sigma), c)$ be the set of natural number tuples

$$\{\langle a_1, \ldots, a_n \rangle \mid \mathcal{D} \models (\exists)((r(\sigma) \wedge c)[x_1/a_1, \ldots, x_n/a_n]\}).$$

The approximation $ap_1(\sigma, c)$ is defined by

$$x_1 \in D_1 \wedge \ldots \wedge x_n \in D_n,$$

and the approximation $ap_2(\sigma, c)$ is defined by

$$x_1 \in \{min_1, \ldots, max_1\} \wedge \ldots \wedge x_n \in \{min_n, \ldots, max_n\},$$

where D_i represents the projection of $dom(r(\sigma), c)$ along its argument i, and min_i and max_i represents the minimum and maximum value in D_i.

Finally, the constraint-solvers for the non-basic constraints are once again based on consistency techniques. However they use the semantics of the constraints to come with an efficient implementation. For instance, inequalities and equations use a reasoning about variation intervals [5, 16]. There can be a large variety of (numeric and symbolic) constraints that might be considered as interesting primitive constraints so that we will not specify the constraint-solvers for them. Note only that the constraint-solvers can be made complete as only domain constraints (i.e relaxation of basic constraints) are considered in conjunction with a non-basic constraint. Also, in [17], we propose a new combinator that makes possible to define most interesting numerical and symbolic constraints from a small set of primitive constraints.

6 Conclusion

This paper has presented an extension to the *Ask & Tell* framework to capture, in a simple and precise way, the operational semantics of CLP languages where some constraints are approximated, inducing an incomplete constraint-solver. The extension consists of two new combinators, relaxed tell and relaxed ask, that are parametrized on a relaxation and approximation function. Constraint are divided into two sets, basic and non-basic constraints. Basic constraints (that can be decided efficiently) are handled as usual while non-basic constraints (that are approximated in terms of basic constraints) are handled through the new combinators.

The extended framework has been instantiated to CLP over finite domains. The basic constraints of CLP(F) have been identified and arc-consistency has been shown to be the basis of an efficient and complete constraint-solver. The relaxation and approximation functions for non-basic constraints in CLP(F) have also been described.

The contributions of this paper include (1) a precise and simple definition of the operational semantics of CLP(F) and (2) the definition of two new combinators that should allow for a precise operational semantics of several other CLP languages.

A structural operational semantics does not describe how to implement the language. What remains to be described for that purpose is how the constraint-solvers are implemented and under which conditions a non-basic constraint is reconsidered after floundering. An efficient arc-consistency algorithm for basic constraints is defined in [4]. The wakening of non-basic constraints is based on a generalization of the techniques used for delay mechanisms (e.g. [11, 2]). Both issues are beyond the scope of this paper.

Future work includes the study of the properties of the above operational semantics as well as its relationships with the declarative and denotational semantics. Application of the approach to other CLP languages with an incomplete constraint-solver will also be considered.

Acknowledgments

Vijay Saraswat gave the initial motivation beyond this paper. His comments were instrumental in showing the value of the SOS semantics. Discussions with Baudouin Le Charlier and the CHIP team of ECRC are also gratefully acknowledged.

References

[1] W Buttner and al. A General Framework for Constraint Handling in Prolog. Technical report, Siemens Technical Report, 1988.

[2] A. Colmerauer. PROLOG II: Manuel de Reference et Modele Theorique. Technical report, GIA - Faculte de Sciences de Luminy, March 1982.

[3] A. Colmerauer. An Introduction to Prolog III. *CACM*, 28(4):412–418, 1990.

[4] Y. Deville and P. Van Hentenryck. An Efficient Arc-Consistency Algorithm for a Class of CSP Problems. Technical Report CS-90-36, CS Department, Brown University, 1990.

[5] M. Dincbas, H. Simonis, and P. Van Hentenryck. Extending Equation Solving and Constraint Handling in Logic Programming. In MCC, editor, *Colloquium on Resolution of Equations in Algebraic Structures (CREAS)*, Texas, May 1987.

[6] M. Dincbas, P. Van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The Constraint Logic Programming Language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, Japan, December 1988.

[7] J. Jaffar and J-L. Lassez. Constraint Logic Programming. In *POPL-87*, Munich, FRG, January 1987.

[8] J. Jaffar and S. Michaylov. Methodology and Implementation of a CLP System. In *Fourth International Conference on Logic Programming*, Melbourne, Australia, May 1987.

[9] A.K. Mackworth. Consistency in Networks of Relations. *AI Journal*, 8(1):99–118, 1977.

[10] R. Mohr and T.C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[11] L. Naish. *Negation and Control in Prolog.* PhD thesis, University of Melbourne, Australia, 1985.

[12] W. Older and A. Vellino. Extending Prolog with Constraint Arithmetics on Real Intervals. In *Canadian Conference on Computer & Electrical Engineering*, Ottawa, 1990.

[13] G.D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, CS Department, University of Aarhus, 1981.

[14] V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, 1989.

[15] P. Van Hentenryck. A Framework for Consistency Techniques in Logic Programming. In *IJCAI-87*, Milan, Italy, August 1987.

[16] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[17] P. Van Hentenryck and Y. Deville. A new Logical Connective and its Application to Constraint Logic Programming. Technical Report CS-90-24, CS Department, Brown University, 1990.

[18] P. Voda. The Constraint Language Trilogy: Semantics and Computations. Technical report, Complete Logic Systems, North Vancouver, BC, Canada, 1988.

[19] C. Walinsky. CLP($\sum^*$). In *Sixth International Conference on Logic Programming*, Lisbon, Portugal, June 1989.

Constructing Hierarchical Solvers for Functional Constraint Satisfaction Problems

Extended Summary

Sunil Mohan

Department of Computer Science,
Rutgers University,
New Brunswick, NJ 08903

mohan@cs.rutgers.edu

This research looks at applying learning techniques for building abstractions that can be used by a hierarchical Constraint Satisfaction Problem solver, of the type described in [1]. There are three main criteria that these abstractions must satisfy:

1. Some aspect of the given problem, the set of constraints, must be evaluable on the abstractions. This will allow portions of the search space to be pruned out at the level of the abstractions.
2. The selected abstractions must be the "best" for the job. There is a large space of abstractions that can be constructed on the given CSP system. Some candidates will perform better in a hierarchical solver than others.
3. The constructed hierarchical solver must be *Hierarchically Complete.* This means that the hierarchical solver must be able to find *all* the solutions. Note that correctness of the solver is also required.

Constraint Satisfaction Problems (CSPs) involve finding values for a fixed number of variables such that a given set of constraints is satisfied. This can be denoted by

$$\text{Find } \{x \in \mathcal{D} | T(x)\}$$

where $\mathcal{D}$ is the search space, and T is the constraint to be satisfied. The hierarchical solver uses an abstraction of the search space where an abstract version of the problem is evaluated, and the abstract solutions then get refined. It can be shown [2, 3] that in a hierarchically complete system, for constraint T with a corresponding abstract problem P, the following relationship must hold:

$$T(x) \rightarrow P(f(x)) \tag{2}$$

where f is the abstraction mapping. Furthermore, if f is used to map the CSP search space into an abstract search space, an implicand of the CSP problem becomes evaluable on those abstractions, thus satisfying criterion 1 above.

A syntactic application of this hierarchical completeness condition to a given CSP can be used to generate candidate abstractions and hierarchical systems. In the first part of our research, we have been investigating such techniques for building a two-level hierarchical solver. This type of solver, similar in principle to ABSTRIPS [4], is essentially built upon two CSP systems: the given original CSP system defining the base level of the hierarchy, and an abstract CSP system defining the single abstract level. Using the hierarchical completeness condition from above, the abstract CSP system can be defined as

$$\text{Find } \{z \in \mathcal{R} \mid P(z)\}$$

where $z = f(x)$ relates the abstract and base level search spaces. The hierarchical solver first finds solutions to P in $\mathcal{R}$, and then searches their refinement in the base level for solutions to T. To enable this refinement process a new constraint, restricting the elements searched to be refinements of the abstract solutions, is added to the base level CSP:

$$\text{Find } \{x \in \mathcal{D} \mid T(x) \wedge (f(x) = z)\}$$

Each level of the hierarchical solver can now be implemented by a chronologically backtracking generate-and-test system.

We have restricted our attention to the class of parameterized Functional Constraint Satisfaction Problems (pFCSP's). A parameterized CSP (pCSP) can be defined as the problem schema

$$\begin{aligned} &\text{Find } \{x \in \mathcal{D} | T(x, y)\} \\ &\quad \text{Where } y \in C \end{aligned}$$

Here y are the problem parameters, and define the problem space. A problem instance is defined by a particular set of values for the problem parameters. A hierarchical system, built using the above techniques, for a pCSP is applicable to all problem instances in the pCSP. The corresponding abstract problem for a given problem instance is derived by simply instantiating the particular version of implication from (2) used to derive the abstract level. A Functional CSP is a CSP whose specification involves the use of function symbols (in the terms).

As a simple example, consider the pCSP shown in figure 1. A generate-and-test solver that finds all the solutions to the problem is shown in figure 2.

$$\begin{aligned} &\text{Find } \{\langle x_1, x_2, x_3, x_4\rangle \in I_{10}^4 | (x_1 * x_2 + x_3 * x_4 = y)\} \\ &\quad \ldots \text{where } I_{10} = \{1, 2, \ldots, 10\}, \text{ and } y \in I_{50} = \{1, 2, \ldots, 50\} \end{aligned}$$

Figure 1: A simple parameterized CSP

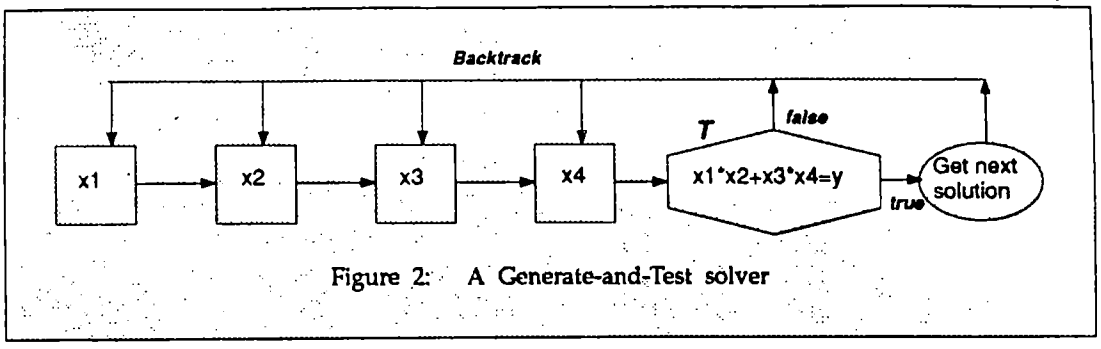


Figure 2: A Generate-and-Test solver

Taking $f = \langle *, * \rangle$ in the example, the constraint $(x_1 * x_2 + x_3 * x_4 = y)$ is decomposed into $P : (z_1 + z_2 = y)$ and $P_1 : (x_1 * x_2 = z_1)$, $P_2 : (x_3 * x_4 = z_2)$. The function $*$ could be specified as $* : \quad I_{10} \times I_{10} \rightarrow C_{100}$ where C_{100} is the set of products from $\{1, \ldots, 10\} * \{1, \ldots, 10\}$ (there are 42 of them). Then $\overline{z} = \langle z_1, z_2 \rangle \in C_{100} \times C_{100}$ becomes the abstract search space. The corresponding hierarchical system is shown in figure 3. If the specified range for $*$ is larger, say $\{1, \ldots, 100\}$, the reduced C_{100} can still be derived using one of the network consistency algorithms described in [5, 6, 7].

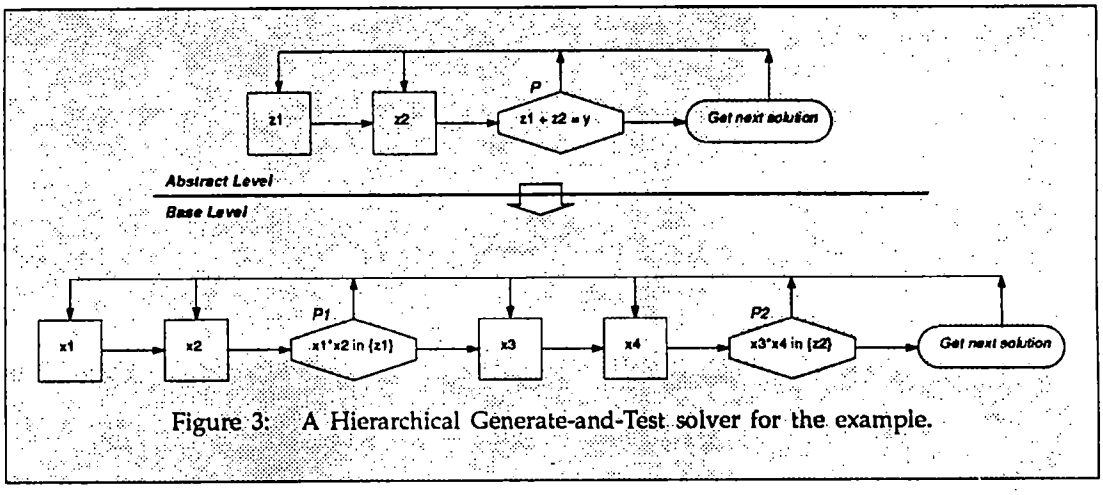


Figure 3: A Hierarchical Generate-and-Test solver for the example.

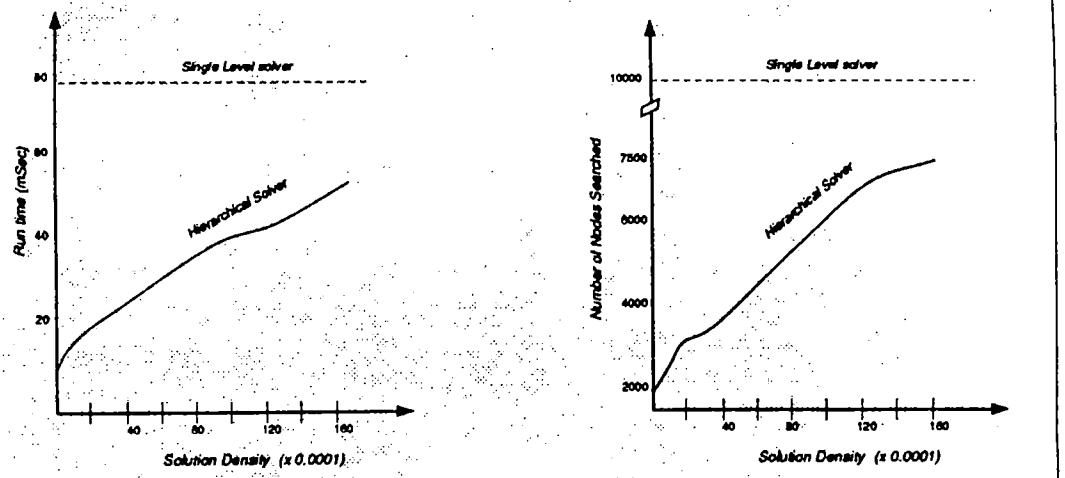
To satisfy criterion 2 stated earlier, we have derived heuristic evaluation functions that give an estimate of the potential of a candidate abstraction for providing a faster hierarchical system [8]. This expression is a function of the ratio of sizes of the two search spaces, called the abstraction ratio, and the solution denstities of the different tests in their domains.

The procedure for building hierarchical CSP solvers has been partially implemented in a research prototype system called HiT (for Hierarchical Transformation). It works on application domains involving function symbols — called Functional pCSPs — and is being tested on toy problems and the domain of floorplanning for buildings. The HiT procedure requires some additional knowledge, beyond the traditional CSP specification, to operate. This includes additional domain knowledge, and data gathered from running experiments.

The space of candidate abstraction functions is defined by the implication in (2). In practice, a partial list of implicands is considered, restricted by a maximum depth of implication. The implicands are generated by applying rules of logic (like modus ponens) to the provided domain knowledge and problem specification. Values for the parameters required by the heuristic evaluation are obtained by running a non-hierarchical solver for the CSP on sample problems. Alternatively, this data can be estimated by using monte-carlo sampling.

The single level and hierarchical generate-and-test systems of figures 2 and 3 were run for several values of the problem parameter. A comparison of their performance for different solution densities is shown in figure 4. The graphs depict run times (on a Sparcsystem 330), and number of nodes (i.e. generator invocations) explored. The single-level system is forced to generate the whole search space because the testing is done only after all the solution parameters have been instantiated. Therefore the run times and nodes expanded for the non-hierarchical system are constant across problem instances. As was expected, the hierarchical system performs much better at lower solution densities.

Another test application used in our experiments is the domain of floorplanning for buildings. A simple floorplanning problem could be of the type "Find all layouts for a house of specified side dimensions, and comprising of a specified number of rooms with constraints on their placement in the house, and on their sizes". Applying the HiT procedure to this problem, we were able to derive two useful abstractions. One abstraction grouped all room sizes and locations on their area. The second abstraction represented a topological view of relative room placements.



250z

Figure 4: A comparison of the performance of the single level and hierarchical solvers for the example problem.

In conclusion, the work reported here is aimed at extending and applying research on concept learning in the machine learning community, with a focus on performance objectives, to the task of building faster hierarchical systems. There has also been interest in building hierarchical solvers in the CSP community. Some researchers [9] have automated construction of abstract CSP systems by relaxing the original CSP network. The abstract CSP system is then used to generate advice on the variable instantiation most likely to lead to a solution. This paper extends that body of research by using reformulation techniques for generating abstract levels, and using the abstraction to reject variable instantiations that do not lead to a solution.

Further work for the near future will be aimed at refining HiT and its evaluation function, and conducting experiments. In the second part of this research, we will be investigating algorithms for building a "component hierarchy" of the type described in [1]. Again, the theme is to extend and apply traditional learning techniques within a performance goal context.

Acknowledgement I would like to thank my advisor Chris Tong, members of the KBSDE project and other colleagues at Rutgers for many helpful discussions and comments.

Bibliography

[1] S. Mittal and F. Frayman, "Making partial choices in constraint reasoning problems," in *proc AAAI*, (Seattle, WA), pp. 631–636, August 1987.

[2] S. Mohan, "Constructing hierarchical solvers for constraint satisfaction problems," AI/Design Working Paper 137, Department of Computer Science, Rutgers University, New Brunswick, NJ, May 1989. Thesis Proposal.

[3] S. Mohan and C. Tong, "Automatic construction of a hierarchical generate-and-test algorithm," in *proc IMLW*, (Ithaca, NY), 1989.

[4] E. D. Sacerdoti, "Planning in a hierarchy of abstraction spaces," *AI*, vol. 5, no. 2, pp. 115–135, 1974.

[5] A. K. Mackworth, "Consistency in networks of relations," *AI*, vol. 8, pp. 99–118, 1977.

[6] E. C. Freuder, "A sufficient condition for backtrack-free search," *JACM*, vol. 29, no. 1, pp. 24–32, 1982.

[7] R. Dechter and J. Pearl, "Network-based heuristics for constraint-satisfaction problems," *AI*, vol. 34, pp. 1–38, 1988.

[8] S. Mohan, "Constructing hierarchical solvers for functional constraint satisfaction problems," AI/Design Project Working Paper 177, Department of Computer Science, Rutgers University, September 1990.

[9] R. Dechter and J. Pearl, "Network-based heuristics for constraint-satisfaction problems," *AI*, vol. 34, pp. 1–38, 1988.

Analysis of Local Consistency in Parallel Constraint Satisfaction Networks (extended abstract)

Simon Kasif Arthur L. Delcher
Department of Computer Science
The Johns Hopkins University
Baltimore, Md 21218

October 22, 1990

1 Introduction

In this paper we present several techniques for parallel processing of constraint networks. One the main underlying assumptions of the Connectionist school of thought is that the connectionist approach is naturally amenable to a parallel (distributed) implementation. Moreover, the claim is often made that the symbolic approach does not have this advantage. Our research is aimed at deriving a precise characterization of the utility of parallelism in constraint networks. In previous papers we demonstrated several methods for parallel execution of special cases of constraint networks such as two label networks and chain networks. In this paper we significantly extend our previous results. We analyze parallel execution for chain networks, tree networks, directed support networks and path-consistency in general networks. While the obvious parallel algorithm for local consistency in constraint networks should work well in practice, we would like to obtain lower and upper bounds on the complexity of the problem on ideal parallel machines (such as the PRAM). This study may also have significant practical implications since it may indicate which parallel primitives are fundamental in the solutions of large constraint systems. Once such primitives are implemented in hardware, they effectively execute in constant time for all practical purposes (e.g., parallel prefix on the Connection Machine). The original design of the Connection Machine was motivated by these considerations. The machine was initially designed to support highly parallel semantic network processing. The ultimate goal of our research is to produce a set of primitives that are critical to the solution of constraint problems.

2 Constraint Satisfaction and Discrete Relaxation

Constraint satisfaction networks are used extensively in many AI applications such as planning, scheduling, natural language analysis and common-sense reasoning (truth

maintenance systems) [dK86, HS79, Mac77, RHZ76, Win84]. These networks use the principle of local constraint propagation to achieve global consistency (*e.g.*, consistent labelling in vision). Below, we give a formal definition of constraint satisfaction networks.

Let $V = \{v_1, \ldots, v_n\}$ be a set of variables. With each variable v_i we associate a set of labels L_i. Now let $\{P_{ij}\}$ be a set of binary predicates that define the compatibility of assigning labels to pairs of variables. Specifically, $P_{ij}(x, y) = 1$ iff the assignment of label x to v_i is compatible with the assignment of label y to v_j.

The Constraint Satisfaction Problem (CSP) is defined as the problem of finding an assignment of labels to the variables that does not violate the constraints given by $\{P_{ij}\}$. More formally, a solution to CSP is a vector $(x_1, \ldots, x_n)$ such that x_i is in L_i and for each i and j, $P_{ij}(x_i, x_j) = 1$.

A standard approach to model CSP problems is by means of a constraint graph. See [Mac77, U.74]. The nodes of the constraint graph correspond to variables of CSP. The edges of the graph correspond to the binary constraints in the CSP. That is, with each edge in the constraint graph we associate a matrix that shows which assignments of labels to the objects connected by that edge are permitted. In this interpretation CSP can be seen as generalized graph coloring.

In this paper we will use an *explicit* constraint graph representation. Given constraint network G we create a new constraint graph that captures the constraints of the initial constraint graph more explicitly.

The construction of an explicit constraint graph is illustrated by example. Assume the set of labels is $\{0, 1\}$. For each edge connecting variables X and Y we create a set of 4 nodes $< X, 0 >$, $< X, 1 >$, $< Y, 0 >$ and $< Y, 1 >$. $< X, L >$ is connected with an arc to $< Y, L' >$ iff assigning L to X assigning L' to Y is consistent. An example is given below.

$< X, 0 >$ $< Y, 0 >$

$< X, 1 >$ $< Y, 1 >$

3 Local Consistency and Discrete Relaxation

Since CSP is known to be $\mathcal{NP}$-complete, several local consistency algorithms have been used extensively to filter out impossible assignments.

Arc Consistency (AC) allows an assignment of a label x to an object s iff for every other object s' in the domain there exists a valid assignment of a label x' which does not violate the constraints. Arc Consistency [Mac77, U.74] is defined formally as follows.

A solution to the local version of CSP (arc consistency) is a vector of sets $(M_1, \ldots, M_n)$ such that M_i is a subset of L_i and a label x is in M_i iff for every M_j, $i \neq j$

there is a y_{xj} in M_j, such that $P_{ij}(x, y_{xj}) = 1$. Intuitively, a label x is assigned to a variable iff for every other variable there is at least one valid assignment of a label to that other variable that supports the assignment of label x to the first variable.

We call a solution $(M_1, \ldots, M_n)$ a *maximal* solution for AC iff there does not exist any other solution $(S_1, \ldots, S_n)$ such that $M_i \subseteq S_i$ for all $1 \leq i \leq n$. We are interested only in maximal solutions for an AC problem. By insisting on maximality we guarantee that we are not losing any possible solutions for the original CSP. Therefore, in the remainder of this paper a solution for an AC problem is identified with a maximal solution. The sequential time complexity of AC is discussed in [MF85]. Discrete relaxation is the most commonly used method to achieve local consistency. Discrete relaxation repeatedly discards labels from variables if the condition specified above (AC) does not hold.

Local consistency belongs to the class of inherently sequential problems called log-space complete for $\mathcal{P}$ (or $\mathcal{P}$-complete). Intuitively, a problem is $\mathcal{P}$-complete iff a logarithmic-time parallel solution (with a polynomial number of processors) for the problem will produce a logarithmic-time parallel solution for every deterministic polynomial-time sequential algorithm. This implies that unless $\mathcal{P} = \mathcal{NC}$ ($\mathcal{NC}$ is the class of problems solvable in logarithmic parallel time with polynomial number of processors) we cannot solve the problem in logarithmic time using a polynomial number of processors.

4 AC in Chains

In a previous paper [Kas89] we observed that a simple separator-based technique can be used to solve constraint satisfaction problems in chains graphs. The technique is based on removing a variable that separates a constraint chain with N variables into two chains with $N/2$ variables. Then we create $2K$ recursive subproblems, where K is the number of labels. We repeat the process $\log N$ times, so that the complexity is $O(K^{\log N})$. For details see [Kas89]. Thus, when the number of labels is large (and becomes a function of the input size, the complexity is exponential, *i.e.*,$O(K^{\log N})$.

Here we propose a simple procedure to achieve AC in a constraint chain. We first construct an explicit constraint graph. The number of nodes in this graph is NK, one node for each variable/label pair. Recall, that two nodes, $< X, a >$ and $< Y, b >$, in an explicit constraint graph are connected iff the assignment of a to X is compatible with assigning b to Y. Without loss of generality, assume the variables are numbered X_1 to X_n in order on the chain. We now orient all the edges in the direction from X_i to X_{i+1}. We mark all nodes reachable from the source nodes containing X_1, and discard all other nodes. Next, we reverse the orientation of edges to point from X_{i+1} to X_i. We then mark all nodes reachable from the nodes in X_n, and discard the rest. We claim that all the nodes that remain correspond to labels that stay.

Theorem 1 *AC in chains is reducible to reachability in directed graphs and therefore solvable with NK^2 processors in $O(\log^2 NK)$ time.*

5 Trees

In the previous paper we addressed the problem of AC in trees. We suggested a separator based algorithm. Similarly to chains, the complexity of that algorithm is $O(\log N)$ with $O(K^{\log N})$ processors. Here we propose a different algorithm. The algorithm operates in $O(\log^2)$ time and uses a polynomial number of processors. More importantly, it reduces the problem to reachability and expression evaluation on trees.

The algorithm is probably inpractical, and is presented to demonstrate the substantial difficulty in getting asympotically optimal parallel algorithms even for tree-like constraint networks.

We sketch the idea of the proof. The formal proof is omitted for length considerations.

Root the tree at any particular node, and construct the explicit graph E. Let $\ell_{i,v}$ denote the $i^{\underline{\text{th}}}$ label in node v, and let $P_{u,v}$ denote the compatibility (support) predicate between nodes u and v. Note that $P_{u,v}$ is also the adjacency matrix for the subgraph of E restricted to labels in nodes u and v.

Step 1: Mark the following set of labels, defined bottom up in the tree: - $\ell_{i,v}$ is marked if v is a leaf. - $\ell_{i,v}$ is marked if $\forall w$ such that w is a child of v $\exists j$ such that $\ell_{j,w}$ is marked and $\ell_{i,v}$ and $\ell_{j,w}$ are adjacent in E. The entire system is consistent iff the set of labels marked at the root is non-empty.

Step 2: Discard all unmarked labels, and consider the original problem restricted only to marked labels. Clear all marks, and do the following second marking procedure: - Mark all labels at the root. - Mark $\ell_{i,v}$ iff $\exists j$ such that $\ell_{j,w}$ is marked and w is the parent of v and $\ell_{i,v}$ and $\ell_{j,w}$ are adjacent in E. The solution is the set of nodes marked at the end of Step 2.

Step 1 can be implemented either as an expression evaluation problem, or by reachability. - As an expression we are computing a bit vector at each node. The operations being performed are multiplication by boolean matrices $P_{u,v}$ and intersection (which is just a bitwise AND). We can evaluate this expression by raking leaves. When a leaf is raked, we are applying an intersection of a known (*i.e.*, constant) bit vector in between 2 matrix multiplications. This is simply a restriction of the matrix product to the rows and columns of the intersection vector. Thus the result of the rake operation can be represented as a single matrix, and can be computed in NC. - By reachability, we can identify all labels reachable from a leaf label (assuming E is now directed with all arcs oriented toward the root). We then mark all labels that are reachable from all leaves under them. We then restrict ourselves only to these marked nodes, and redo the reachability from leaves calculation. (the formal proof can be obtained by induction on the level in the tree).

Step 2 is simply reachability.

Theorem 2 *AC is $\mathcal{NC}$ in trees for arbitrary number of labels.*

6 Directed Arc Consistency (DAC)

Traditionally, AC was considered as a prefiltering step in order to solve constraint satisfaction problems. However, there is an interesting modal-logic-like interpretation of AC in the context of support networks. We start with a collection of agents (variables) each holding a set of beliefs (labels). For each pair of variables X and Y we define a directional support relation that for each belief states which beliefs at one agent get supported by beliefs at the other agents.

An agent keeps a belief a if it is supported by at least one belief at each of the other agents. Note, that we are not seeking a global interpretation of the variables but rather would like to filter unsupported beliefs. This is analogous to the standard AC problem, however, the support relations are directed. Judea Pearl communicated to us several applications of directed support networks. David McAllester pointed out that this version of local consistency is not useful in the context of constraint satisfaction problems where one is seeking to find single assignments for variables. Directed AC is naturally described by an explicit constraint graph. Edges in the graph describe directed support (see example below). The only label that drops is E from Y.

$$\begin{array}{ll} < X, B > & < Y, E > \\ & < Y, D > \\ & < Y, C > \\ < X, A > & < Y, B > \end{array}$$

The following surprising result indicates that the parallel complexity of solving directed AC is considerably more complicated than the standard AC problem. Essentially, this result states the parallel complexity is dependent on the structure of the explicit constraint graph, rather than the structure of the constraint graph. This follows from the fact that we can encode general logical dependency in such a graph.

Theorem 3 *Directed Arc Consistency (DAC) is P-complete, even if the underlying graph is a 3-node chain.*

Proof: Our reduction is from Propositional Horn-Clause Solvability. Without loss of generality we assume the Horn clause program contains a single assertion T, and that every other variable A_i appears as the head of either exactly one rule of the form $A_i \leftarrow A_j, A_k$, or else two or more rules of the form $A_i \leftarrow A_j$. Let A_n be the goal of the program.

We construct an instance of DAC containing three nodes: L(eft), M(iddle) and R(ight). Let each of these three nodes contain a label for T and for each A_i. Now construct arcs for the explicit support graph as follows:

1. From each label in M to the corresponding label in both L and R.
2. From T_L to T_M and from T_R to T_M.
3. For every clause of the form $A_i \leftarrow A_j, A_k$, construct an arc from $A_{j,L}$ to $A_{i,M}$ and from $A_{k,R}$ to $A_{i,M}$.
4. For every clause of the form $A_i \leftarrow A_j$, construct an arc from $A_{j,L}$ to $A_{i,M}$ and from T_R to $A_{i,M}$.

It is now easy to show that the labels that remain in M correspond exactly to the variables that are true in the Horn-clause program. In particular $A_{n,M}$ remains iff the goal A_n of the program is true. □

7 Path Consistency

In this section we make a simple observation about path consistency. Parallel path consistency is also studied in [LR90]. In [Kas85, Kas89] we provided a two way reduction between arc consistency and propositional Horn satisfiability (PHS). Here, we observe that a similar two way reduction can be provided between path consistency and PHS. We illustrate the reduction from 3-consistency to PHS. Without loss of generality assume all labels are unique (otherwise, we can rename the labels). For each pair of labels A and B we create a proposition $P < A, B >$ that is true iff the pair (A, B) gets dropped from consideration. However, $P < A, B >$ is true iff it cannot be extended to at least one variable. We denote this condition by a predicate $P < A, B, i >$ which is true iff this assignment cannot be extended to variable X_i. Thus,

$$\begin{aligned} P < A, B > &\leftarrow P < A, B, 1 >. \\ P < A, B > &\leftarrow P < A, B, 2 >. \\ P < A, B > &\leftarrow P < A, B, 3 >. \\ &\ldots \end{aligned}$$

Now, $P < A, B, i >$ is true iff $P < B, C >$ or $P < A, C >$ is true for all labels C that potentially support the assignment $< A, B >$ at variable X_i. Thus,

$$\begin{aligned} P < A, B, 1 > &\leftarrow P < A, B, C1 >, P < A, B, C2 >, ... \\ P < A, B, C1 > &\leftarrow P < A, C1 >. \\ P < A, B, C1 > &\leftarrow P < B, C1 >. \\ &\ldots \end{aligned}$$

We have $O((EK^2))$ predicates of the form $P < A, B >$. Therefore, the size of the input is dominated by the clauses of the form
$P < A, B, 1 > \leftarrow P < A, B, C1 >, P < A, B, C2 >, ...$ The total size is therefore $O(EK^2(NK)) = O((NEK^3))$ or alternatively $O((N^3K^3))$. The assertions of the program are obtained by applying 3-consistency check once, and asserting all the dropped pairs $< A, B >$ as facts $P < A, B >$.

The important corollary of this construction is that now we can utilize the standard linear time PHS algorithm for local consistency. This immediately yields an algorithm

that matches (in asymptotic complexity) the best known algorithm for path consistency. This algorithm is optimal if the number of labels is assumed constant. The same algorithm would also work for k-consistency with similar optimal performance. The bottom-up PHS algorithm also has very good performance in practice. The conversion to PHS can be done efficiently in parallel.

Similar observations were made independently by McAllester (Ph.D. thesis), Bible, Mackworth and Reiter (previous KR conference), and Saraswat (workshop on constraint systems 1990). Here we presented a careful analys of the transformation that leads to optimal complexity results as stated above.

8 Summary of Parallel AC Results

In this section we summarize our knowledge of parallel complexity of local consistency problems in constraint satisfaction problems. The results appear in the tables below. We classify the problems according to their parallel complexity into two classes: $\mathcal{P}$-complete problems and $\mathcal{NC}$ problems. $\mathcal{P}$-complete are perceived to be difficult to parallelize (in the same sense NP-complete problems are considered intractable), and $\mathcal{NC}$ problems can be solved in polylogarithmic time with a polynomial number of processors. $\mathcal{NC}$-problems are often amenable for optimal speed-up on parallel machines. R denotes relation in the tables.

Acknowledgements This research has been partially supported by the Air Force Office of Scientific Research under grant AFOSR-89-1151 and National Science Foundation under grant IRI-88-09324. Thanks are due to David McAllester, Judea Pearl and Rina Dechter for constructive comments.

Table 1: Complexity of Arc Consistency for Arbitrary-Size Label Sets

Arbitrary K (K is the size of the label set L)		
G	CSP	AC
Chain	$\mathcal{NC}$; reachability	Undirected R's: $\mathcal{NC}$; reachability Directed R's: P-complete; reduction from Propositional Horn-Clause Solvability
Tree	$\mathcal{NC}$;	Undirected R's: $\mathcal{NC}$; like expression eval where operation at each node is intersection of sets of support for each label (see this paper) Directed R's: P-complete; from above
Simple Cycle	$\mathcal{NC}$; reachability	Undirected R's: $\mathcal{NC}$; cycle detection Directed R's: P-complete; from above
Arbitrary Graph	$\mathcal{NP}$-complete; reduction from graph colouring	Undirected R's: P-complete; reduction from Propositional Horn-Clause Solvability Directed R's: P-complete; from above

Table 2: Complexity of Arc Consistency for Fixed-Size Label Sets

Fixed K (K is the size of the label set L)		
G	CSP	AC
Chain	$\mathcal{NC}$	Undirected R's: $\mathcal{NC}$ Directed R's: $\mathcal{NC}$;
Tree	$\mathcal{NC}$	Undirected R's: $\mathcal{NC}$ Directed R's: $\mathcal{NC}$; from above
Simple Cycle	$\mathcal{NC}$	Undirected R's: $\mathcal{NC}$ Directed R's: $\mathcal{NC}$; from above
Arbitrary Graph	$K = 2$: Linear sequential algorithm by reduction to 2-SAT which is $\mathcal{NC}$ $K \geq 3$: $\mathcal{NP}$-complete; reduction from 3-colouring graphs	Undirected R's: For $K = 2$, $\mathcal{NC}$ by reachability along "singleton paths"; For $K \geq 3$, P-complete from Propositional Horn-Clause Solvability Directed R's: P-complete for $K \geq 2$

References

[dK86] J. de Kleer. An assumption-based tms. *Artificial Intelligence*, 28:127–162, 1986.

[HS79] R. M. Haralick and L. G. Shapiro. The consistent labeling problem: Part I. *IEEE Trans. Patt. Anal. Mach. Intel.*, PAMI-1:173–184, 1979.

[Kas85] S. Kasif. On the parallel complexity of discrete relaxation. Technical Report JHU/EECS-85/18, Johns Hopkins University, 1985. (AI Journal 1990).

[Kas89] S. Kasif. Parallel solutions to constraint satisfaction problems. In *Principles of Knowledge Representation and Reasoning*, May 1989.

[LR90] P. Ladkin and Maddux R. Parallel path consistency algorithms for constraint satisfaction. In *Proceedings of the 1990 Workshop on Constraint Systems*, August 1990.

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

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

[RHZ76] A. Rosenfeld, R. Hummel, and S. Zucker. Scene labeling by relaxation operations. *IEEE Trans. Syst. Man Cybern.*, SMC-6:420–433, 1976.

[U.74] Montanari U. Networks of constraints: Fundamental properties. *Inform. Sci.*, 7:727–732, 1974.

[Win84] P. H. Winston. *Artificial Intelligence.* Addison-Wesley, 1984.

Parallel Path Consistency

Steven Y. Susswein, Thomas C. Henderson and Joe Zachary

Department of Computer Science
University of Utah
Salt Lake City, UT 84112

Abstract

Filtering algorithms are well accepted as a means of speeding up the solution of the consistent labeling problem (CLP). Despite the fact that path consistency does a better job of filtering than arc consistency, AC is still the preferred technique because it has a much lower time complexity.

We are implementing parallel path consistency algorithms on a multiprocessor and comparing their performance to the best sequential and parallel arc consistency algorithms. We also intend to categorize the relation between graph structure and algorithm performance. Preliminary work has shown linear performance increases for parallelized path consistency and also shown that in many cases performance is significantly better than the theoretical worst case. These two results lead us to believe that parallel path consistency may be a superior filtering technique. Moreover, we conjecture that no set of relations exist of n nodes and m labels which requires more than mn iterations of Path Consistency to make the relations consistent.

1 Introduction

There is a class of problems in computer science known variously as *Consistent Labeling Problems* [4], *Satisfycing Assignment Problems* [2], *Constraint Satisfaction Problems* [6], etc. We will refer to it as the *Consistent Labeling Problem* (CLP). Many classical computer science problems such as N-queens, magic squares, and the four color map problem can be viewed as *Consistent Labeling Problems*, along with a number of current problems in computer vision.

The basic problem can be looked at abstractly in the form of a graph, in which we have:

- A set of *nodes*, N = $\{n_1, n_2, \ldots, n_n\}$; (let $|N| = n$).
- For each node n_i a domain M_i, which is the set of acceptable labels for that node. Often all the M_i's are the same, giving $M_1 = M_2 = \ldots = M_n = M$; (let $|M| = m$).
- A set of *constraint relations* R_{ij}, $i, j = 1, n$, which define the consistent label pairs which can be assigned to nodes n_i and n_j; i.e., $R_{ij}(l_1, l_2)$ means label l_1 at node n_i with label l_2 at node n_j is a consistent labeling. Directed arcs give a visual representation of the relationships.

The problem is to find a complete and consistent labeling such that each node is assigned a label from its label set that satisfies the constraints induced by all its connected arcs.

For example, take a three node graph where the arcs represent the relationship "equals." That is, the labels assigned to the two nodes at the ends of each arc must be equal. If the label sets for the nodes are:

- $node_1$: {1,2,3}
- $node_2$: {2,3,4}
- $node_3$: {3,4,5}

then the only possible solution is:
$node_1 = 3$
$node_2 = 3$
$node_3 = 3$

1.1 Solutions to CLP

It can be shown that CLP is NP-complete[3]. Thus there are no known efficient solutions. However, there are a number of ways the problem can be solved, including *generate and test*, *standard backtracking*, *Waltz filtering*[11], etc. In standard backtracking, we assign a label to $node_1$, and using this constraint attempt to find a valid label for $node_2$. Using these values for nodes one and two, we attempt to find a valid label for $node_3$, etc. When no valid label

exists for a node, we backtrack and make a new assignment for the last node. We continue until all nodes have been assigned labels or all possible assignments have been attempted, and failed.

Mackworth [7] has shown that the "thrashing" behavior of standard backtracking can be reduced by the incorporation of consistency algorithms (*node, arc,* and *path* consistency). Mohr and Henderson [8] have given an optimal algorithm for arc consistency and an improved algorithm for path consistency.

- In *node consistency*, we look at the label set for a single node and remove any impossible labels.
- In *arc consistency* we look at each pair of nodes and remove those labels which cannot satisfy the arc between them. For example, if we looked at nodes one and two in the above example using arc consistency we would remove the value 1 from $node_1$ and the value 4 from $node_2$.
- In *path consistency* we look at groups of three or more nodes (Montanari has shown that if all paths of length two are consistent then the entire graph is consistent, so we actually look at paths of length exactly two).

Path consistency does a much better job of filtering than arc consistency, but is also much slower (i.e., requires a lot more computation); as a result, arc consistency is currently the most widely used filtering technique.

1.2 Parallel Algorithms for AC and PC

Samal has explored parallel versions of arc consistency [10]. He showed that the worst case performance of any parallel arc consistency algorithm is $O(mn)$. This means that given a polynomial bound on the number of processors, it takes time proportional to mn to solve the problem in the worst case. Moreover, he explored the dependence of performance on graph structure.

We are interested in providing a similar analysis for parallel path consistency algorithms. We conjecture that the average case time complexity of parallel path consistency is $O(mn)$. This means that over populations of standard problems and given a polynomial bound on the number of processors, the average time to solve the path consistency problem is proportional to mn. In fact, our preliminary results indicate that the innermost loops of path consistency (i.e., those which update the relations) run in constant time, $O(1)$. We therefore propose the following conjecture:

Linearity of Parallel Path Consistency: No set of relations R_{ij}, $i, j = 1, n$, exists which requires more than mn iterations of Parallel Path Consistency to make them consistent.

2 Parallel Path Consistency

The current best path consistency algorithm (PC-3) has a time complexity of $O(n^3m^3)$, compared to the optimal arc consistency algorithm (AC-4) which has a time complexity of $O(n^2m^2)$[5], but path consistency does a much better job of pruning the search space. This can be seen by looking at the *4-Queens* problem. Path consistency will prune 50% of the labels from each node, leaving just two possible positions for each queen; arc consistency on the other hand prunes *none* of the labels, leaving the problem at its original complexity.

The main thrust of this research is to define and implement *parallel* versions of the PC algorithms on a multiprocessor to see whether they can outperform the best AC algorithms when used within search to prune the search tree at each node.

2.1 Standalone Parallel PC

We are currently investigating parallel versions of the PC algorithms and comparing their performance to each other and to the parallel AC algorithms measured by Samal. Samal has shown that the best sequential AC algorithm is not necessarily the best parallel algorithm. For each algorithm we will measure its raw speed as well as its speedup linearity, with the goal of finding a parallel PC algorithm with at least linear speedup. Speedup linearity is a measure of how well we are utilizing the additional processors and is defined as *time on 1 processor/ (N × time on N processors).*

2.2 Using PC in Search

The next step involves creating a standard backtracking program, in which various parallel AC and PC routines are embedded. At each node of the search tree we run the chosen AC or PC code to check for consistency. Again, we are measuring the raw performance and speedup, as well as the average, minimum, and maximum search depth and the number of nodes traversed.

2.3 Finding Worst Case Performance

Although theoretical worst case performance of sequential PC-1 is of complexity $O(m^5n^5)$, early experiments have shown actual performance to be much better (see section 3.3). We are attempting to find and categorize the worst case performance based on the type of graph and constraint relation.

N-queens and confused n-queens[9] are the standard test cases for performance measurement and comparison.

3 Initial Results

We have conducted some simple experiments. These experiments support the following claims:

1. *Path consistency* prunes the search space to a greater extent than *arc consistency.*
2. Highly parallelized versions of *path consistency* can achieve near-linear speedup.
3. *Path consistency* will normally run in much better than theoretical worst case performance.

3.1 Pruning Efficiency of PC vs. AC

We already had a working version of arc consistency created by Samal, so PC-1 was coded based on the algorithm given by Mackworth. Both these programs use identical system calls to report timing information and were run on a number of both consistent and inconsistent graphs. These graphs mostly corresponded to the *N-Queens* problem (for various values of N), but other graphs were also examined. As expected, arc consistency ran much faster than path consistency, but path consistency did a superior job of pruning the search space. As mentioned earlier, a good example of this is consistent 4-Queens. Figure 1 shows number of nodes expanded for n-queens ($n = 4, 6, 8, 10$).

3.2 Parallel PC-1

As a next step, we modified the PC-1 program mentioned above to run as a parallel program on the Butterfly. We employed a straightforward parallelization, where the number of parallel processes generated is based on the size of the initial graph. Larger graphs have shown an approximately linear speedup, up to the number of processors available (see Table 1). Note that the number of iterations varies slightly due to interactions caused by the parallelization, and the speedup remains linear only for equal iteration counts.

3.3 Worst Case Performance

The graph input to PC-1 is encoded in the form of an $nm*nm$ binary matrix. The algorithm iterates over this matrix until two successive iterations yield no change in the matrix. Each iteration is of complexity $O(m^3n^3)$ and can only simplify the matrix (i.e., change a "1" to a "0"). Since each iteration simplifies at least one element in the matrix, we require as a worst case m^2n^2 iterations, yielding a worst case performance of $O(m^5n^5)$.

Since the input matrix defines both the list of possible labels for each node *and* the constraint relation between nodes, it is possible to exhaustively examine all possible relation

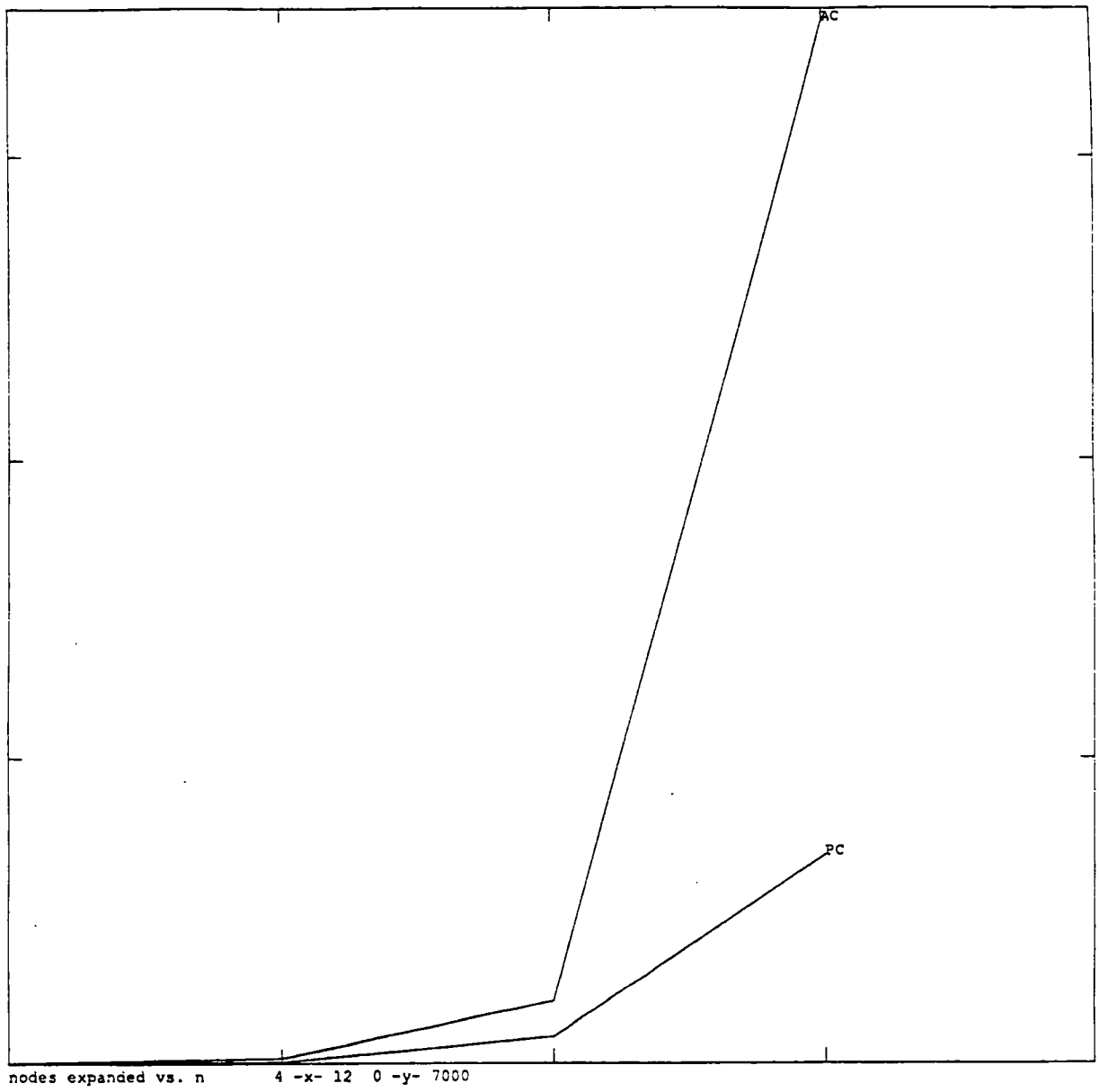


Figure 1: Number of Nodes Expanded in N-Queens for AC and PC

processors	*raw time (ms)*	*iterations*	*speedup linearity*	
			2 iterations	*3 iterations*
1s	602980	2	1.00	
1p	626305	2	0.96	
2	322272	2	0.94	
3	213479	2	0.94	
4	244888	3		0.62
5	128830	2	0.94	
6	169108	3		0.59
7	142597	3		0.60
8	123289	3		0.61
9	113087	3		0.59
10	101053	3		0.60
11	93206	3		0.59
12	84602	3		0.59
13	78071	3		0.59
14	72184	3		0.60
15	44201	2	0.91	

Note: 1s is sequential code and 1p is parallel code

Table 1: Speedup Linearity for 16-Queens using PC-1

constraints for small values of m and n through a brute-force approach of constructing all possible input matrixes. The purpose of this experiment was to find which constraint relations produced the worst results (greatest number of iterations). While we haven't been able to fully characterize which constraint relations produced the most iterations, we were surprised by the maximum and average number of iterations required. Using values of $m = 2$ and $n = 3$ yielded a worst case performance of 5 iterations (compared to a theoretical worst case of 30 iterations) and an average case performance of 2.07 iterations (see Figure 2).

Additional experiments varying the value of m and n for a fixed relation showed that the number of iterations required remains small and relatively constant for at least some relations. If found to be generally true for all relations this would make parallel path consistency even more attractive. Each iteration in PC-1 can be highly parallelized, but the iterations themselves are performed in sequence. The number of iterations required (whose upper bound is theoretically m^2n^2) places an upper bound on the efficiency of parallel PC; if the number of iterations required is found to be small and relatively constant for large values of m and n, then parallel path consistency may prove to be a superior filtering technique.

4 Tools and Facilities

All the code is being written in standard *C*. Timing information is gathered using standard *Unix* system calls (for the sequential code) and *Uniform* built-in timing routines (for the parallel code).

4.1 DECStation 3100

Sequential code is developed and run on a dedicated DECStation 3100, a high-performance RISC workstation. Code developed here under *ULTRIX* is source-code compatible with the University Bobcat workstations, but its high performance (approximately 3× an HP370) and lack of contending jobs means that large runs can be completed quickly.

4.2 Butterfly GP1000

Parallel code is being developed and run on the BBN Butterfly multiprocessor. The Butterfly offers two means of accessing its multiprocessor features: direct system calls to the *Mach* operating system, and the *Uniform* system. The *Uniform* system consists of a library of routines which allow easy access to the multiprocessing features. While not as powerful as direct *Mach* calls, it is much easier to use and supplies all the features needed to implement parallel path consistency.

The Butterfly is configured with eighteen nodes, which will be sufficient for development and testing and to show the effect of parallelized PC, but we also hope to gain access to a

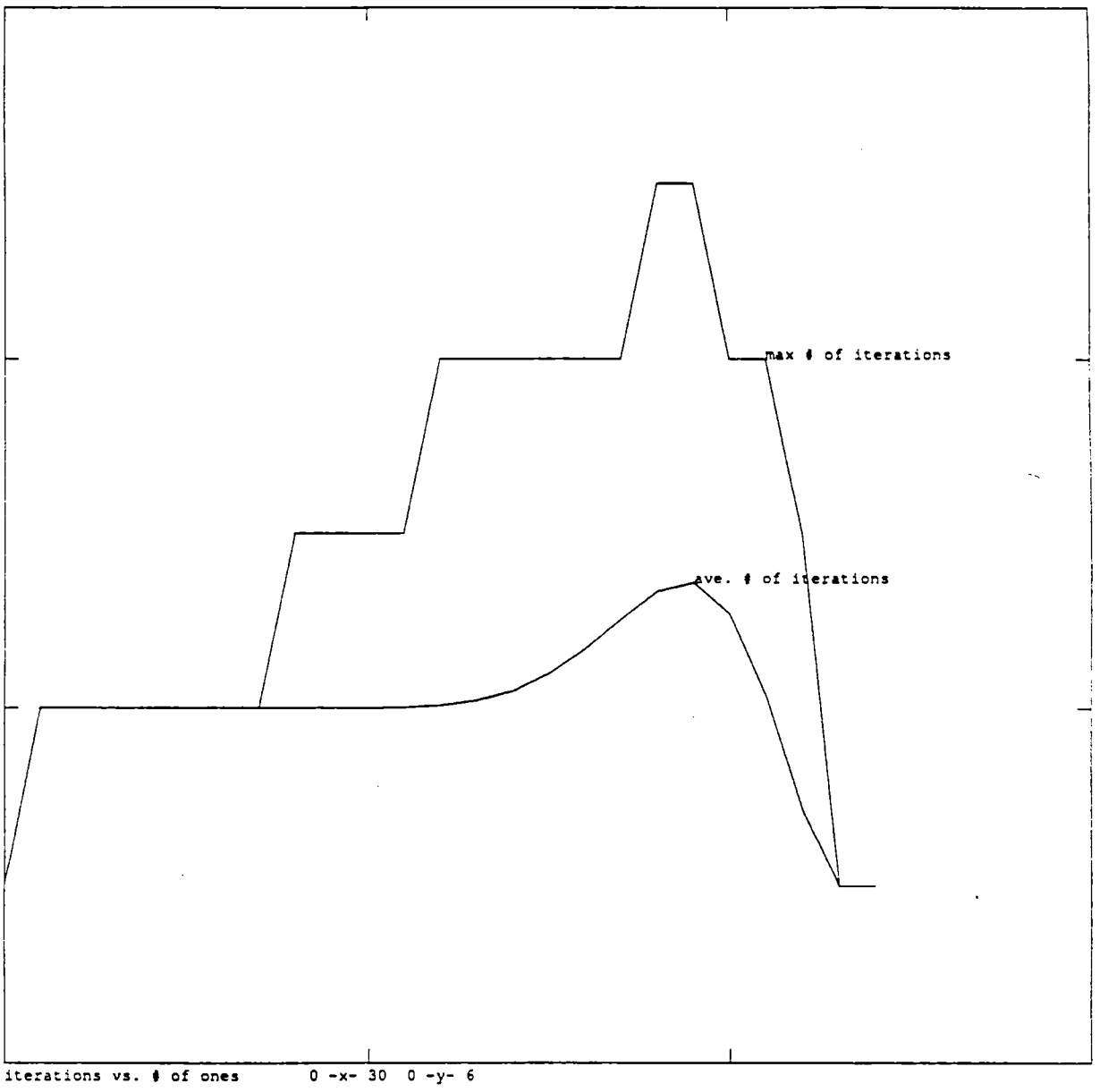


Figure 2: Avg. and Max. Iterations for all relations of $m = 3$ and $n = 2$.

40 node Butterfly located at Cornell University to verify that my results hold for a larger degree of parallelization.

4.3 Connection Machine

In future work, we plan to represent and compute Path Consistency as an outer product [1] on a fine grain connection machine at Los Alamos. We hope to determine from this whether the speedup is sufficient to motivate the investigation of a special-purpose integrated circuit.

References

[1] S. Esener G. Marsden, F. Kiamilev and S. Lee. *Highly Parallel Algorithms for Constraint Satisfaction.* Technical Report, UCSD, 1990.

[2] John Gaschnig. *Performance Measurement and Analysis of Certain Search Algorithms.* Technical Report CMU-CS-79-124, Carnegie-Mellon University, May 1979.

[3] R. Haralick, L. Davis, A. Rosenfeld, and D. Milgram. Reduction Operations for Constraint Satisfaction. *Information Sciences*, 14:199–219, 1978.

[4] R. Haralick and L. Shapiro. The Consistent Labeling Problem. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):173–183, April 1979.

[5] T. Henderson. *Discrete Relaxation Techniques.* Oxford University Press, New York, 1990.

[6] A.K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[7] A.K. Mackworth and E.C. Freuder. The Complexity of Some Polynomial Network Consistency Algorithms for Constraint Satisfaction Problems. *Artificial Intelligence*, 25:65–74, 1985.

[8] Roger Mohr and Thomas C. Henderson. Arc and Path Consistency Revisitied. *Artificial Intelligence*, 28(2):225–233, March 1986.

[9] B. Nadel. Constraint Satisfaction Algorithms. *Computational Intelligence*, 5(4):188–224, november 1989.

[10] A.K. Samal. *Parallel Split-Level Relaxation.* PhD thesis, University of Utah, Salt Lake City, Utah, August 1988.

[11] D. Waltz. Understanding Line Drawings of Scenes with Shadows. In ed. P.H. Winston, editor, *The Psychology of Computer Vision*, pages 19–91, McGraw-Hill, New York, 1975.

A Distributed Solution to the Network Consistency Problem

Zeev Collin
Computer Science Department
Technion – Israel Institute of Technology
Haifa, Israel 32000

Rina Dechter (1)
University of California
Information and Computer Science
Irvine, CA 92717

Abstract

In this paper we present a distributed algorithm for solving the binary constraint satisfaction problem. Unlike approaches based on connectionist type architectures, our protocol is guaranteed to be self stabilized, namely it converges to a consistent solution, if such exists, from any initial configuration. An important quality of this protocol is that it is self-stabilizing - a property, that renders our method suitable for dynamic or error prone environments.

1. INTRODUCTION

Consider the distributed version of the graph coloring problem, where each processor must select a color (from a given set of colors) that is different from any color selected by its neighbors. This coloring task, whose sequential version (i.e. graph coloring) is known to be NP-complete, belongs to a large class of combinatorial problems known as **Constraint Satisfaction Problems (CSPs)** which present interesting challenges to distributed computation, particularly to connectionist architectures. We call the distributed version of the problem the **network consistency problem.** Since the problem is inherently intractable, the interesting questions for distributed models are those of feasibility rather than efficiency. The main question we wish to answer in this paper is: What types of distributed models would admit a self-stabilizing algorithm, namely, one that converges to a solution, if such exists, from any initial state of the network.

The motivation for addressing this question stems from attempting to solve constraint satisfaction problems within a "connectionist" type architecture. Constraints are useful in programming languages, simulation packages and general knowledge representation systems because they permit the user to state declaratively those relations that are to be maintained, rather than writing the procedures for maintaining the relations. The prospects of solving such problems by connectionist networks promise the combined advantages of massive parallelism and simplicity of design. Indeed, many interesting problems attacked by neural networks researchers involve constraint satisfaction [1, 16, 3], and, in fact, any discrete state connectionist network can be viewed as a type of constraint network, with each stable pattern of states representing a consistent solution. However, whereas current connectionist

(1) This research was supported in part by NSF grant #IRI-8815522 and by Air Force grant #AFSOR 80-0136 while the second author was visiting the cognitive systems lab at UCLA.

approaches to CSPs lack theoretical guarantees of convergence (to a solution satisfying all constraints), the distributed model which we use here is the closest in spirit to the connectionist paradigm for which such guarantees have been established. Other related attempts for solving CSPs distributedly were either restricted to singly connected networks [4, 15], or, were based on a general constraint propagation, thus not guarantee convergence to a consistent solution [12].

Our distributed model consists of a network of interconnected processors in which an activated processor reads the states of all its neighbors, decides whether to change its state and then moves to a new state. The activation of a processor is determined by its current state and the states of its neighbors. We also assume that all processors but one are identical (this assumption is not part of classical connectionist models, but is necessary for our protocol). Under these architectural restrictions the network consistency problem is formulated as follows: Each processor has a pre-determined set of values and a compatibility relation indicating which of its neighbors' values are compatible with each of its own. Each processor must select a value that is compatible with the values selected by its neighbors. We shall first review the sequential variant of this problem and then develop a distributed self-stabilizing solution.

A network of binary constraints involves a set of n variables $X_1,...,X_n$, each represented by its domain values, $D_1, \ldots, D_n$, and a set of constraints. A **binary constraint** R_{ij} between two variables X_i and X_j is a subset of the cartesian product $D_i \times D_j$ that specifies which values of the variables are compatible with each other. A solution is an assignment of values to all the variables which satisfies all the constraints, and the **constraint satisfaction problems (CSP)** associated with these networks are to find one or all solutions. A binary CSP can be associated with a **constraint-graph** in which nodes represent variables and arcs connect pairs of variables which are constrained explicitly. Figure 1a presents a constraint network where each node represents a variable having values $\{a, b, c\}$ and each link is associated with a strict lexicographic order (where $X_i < X_j$ iff $i < j$). (The domains and the constraints are explicitly indicated on some of the links.)

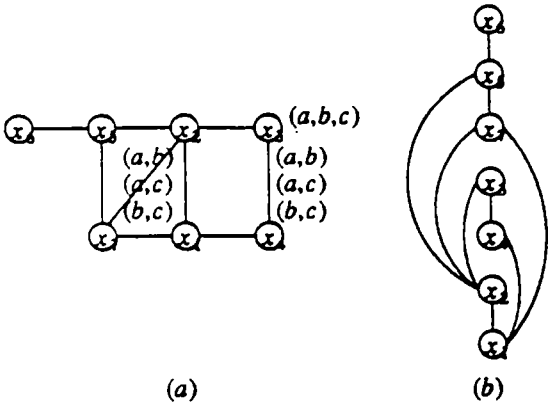


Figure 1: An example of a binary CN

General constraint satisfaction problems may involve constraints of any arity, but since network communication is only pairwise we focus on this subclass of problems.

The rest of this paper is organized as follows: Section 2 provides the sequential algorithm for solving a CSP, that is the basis for our distributed protocol. Section 3 introduces the distributed model and the requirements for self-stabilization, section 4 provides the self-stabilizing distributed protocol for solving the network consistency problem, while section 5 presents some worst-case analysis of the performance of our protocol.

2. SEQUENTIAL ALGORITHMS FOR CONSTRAINT SATISFACTION

2.1 Backtracking

The most common algorithm for solving a CSP is **backtracking**. In its standard version, the algorithm traverses the variables in a predetermined order, provisionally assigning consistent values to a subsequence $(X_1, \ldots, X_i)$ of variables and attempting to append to it a new instantiation of X_{i+1} such that the whole set is consistent. If no consistent assignment can be found for the next variable X_{i+1}, a deadend situation occurs; the algorithm "backtracks" to the most recent variable, changes its assignment and continues from there. A backtracking algorithm for finding one solution is given below. It is defined by two recursive procedures, **Forward** and **Backword**. The first extends a current partial assignment, if possible, and the second handles deadend situations. The procedures maintain lists of candidate values (C_i) for each variable X_i.

```
Forward ( x_1, . . . , x_i)
Begin
  1. if i = n exit with the current assignment.
  2. C_{i+1} ← Compute-candidates(x_1, . . . ,x_i,X_{i+1})
  3. if C_{i+1} is not empty then
  4.    x_{i+1} ← first element in C_{i+1}, and
  5.    remove x_{i+1} from C_{i+1}, and
  6.    Forward(x_1, . . . ,x_i,x_{i+1})
  7. else
  8.    Backword(x_1, . . . ,x_i)
End.

Backword( x_1, . . . ,x_i)
Begin
  1. if i =0, exit   { No solution exists }
  2. if C_i is not empty then
  3.    x_i ← first in C_i, and
  4.    remove x_i from C_i, and
  5.    Forward(x_1, . . . ,x_i)
  6. else
  7.    Backword(x_1, . . . ,x_{i-1})
End.
```

The procedure **compute-candidates**$(x_1, \ldots, x_i, X_{i+1})$ selects all values in the domain of X_{i+1} which are consistent with the previous assignments.

2.2 Backjumping

Many enhancement schemes were proposed to overcome the inefficiency of "naive" backtracking [14, 13, 9, 11]. One particularly useful technique, called **backjumping** [5] consults the topology of the constraint graph to guide its "backword" phase. Specifically, instead of going back to the most recent variable instantiated it **jumps back** several levels to the first variable **connected** to the deadend variable.

Consider again the problem in figure 1a. If variables are instantiated in the order $X_1, X_2, X_4, X_3, X_7, X_5, X_6$ (see figure 1b), then when a dead-end occurs at X_7 the algorithm will jump back to variable X_2; since X_7 is not connected to either X_3 or X_4 they cannot be responsible for the deadend. If the variable to which the algorithm retreats has no more values, it backs-up further, to the most recent variable connected either to the original or to the new deadend variables, and so on.

2.3 Depth first search with backjumping

Whereas the implementation of backjumping in an arbitrary variable ordering requires a careful maintenance of each variable's parents set [5], some orderings facilitate a specially simple implementation. If we use a depth-first search (DFS) on the constraint graph (to generate a DFS tree) and then conduct backjumping in an inorder traversal of the DFS tree [8], finding the jump-back destination amounts by following a very simple rule: if a deadend occurred at variable X, go back to the parent of X in the DFS tree. Consider once again our example of figure 1. A DFS tree of this graph is given in figure 2, and an inorder traversal of this tree is $(X_1, X_2, X_3, X_4, X_5, X_6, X_7)$. Hence, if a deadend occured at node X_5 the algorithm retreats to its parent, X_2.

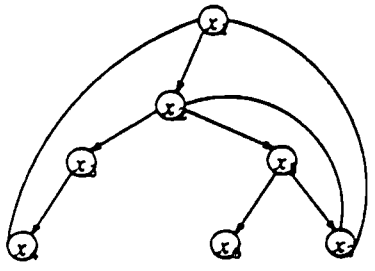


Figure 2 : A DFS tree

The nice property of a DFS tree, which makes it particularly suitable for parallel implementation, is that any arc of the graph, which is not in the tree, connects a node to one of its tree ancestors (i.e. along the path leading to it from the root). Namely, the DFS tree represents a useful decomposition of the graph: if a variable X and all its ancestors are removed from the graph, the subtrees rooted at X will be disconnected. This translates to a useful problem-decomposition strategy: if all ancestors of variable X are instantiated, then the solutions of all its subtrees are completely independent and can be performed in parallel. The idea of using a DFS tree traversal for backtracking and its potential for parallel implementation is not new. It was introduced by Freuder and Quinn [11]. However, the parallel algorithm they present assumes a message passing model, it is targeted for implementation

on a multiprocessor and it is not self-stabilizing [10]. We believe that the use of a DFS-based backjumping for a connectionist type architecture and its self-stabilizing property is novel to this work.

3. BASIC DEFINITIONS FOR DISTRIBUTED COMPUTATIONS

3.1 The model

Our general communication model is similar to the one defined in [7]. A distributed system consists of n processors, $P_0, P_1, \cdots P_{n-1}$, connected by bidirectional communication links. It can be viewed as a **communication graph** where nodes represent processors and arcs correspond to communication links. We use the terms **node** and **processor** interchangeably. Some (or all) edges of the graph may be directed, meaning that the two linked processors (called a **child** and a **parent** respectively), are aware of this direction (the link directions, though, are unrelated to the communication flow). Neighbors communicate using shared communication registers, called **state registers**, and $state_i$ is the register written only by node i, but may be read by several processors (all i's neighbors). The state register may have a few fields, but it is regarded as one unit. This method of communication is known as **shared memory multi-reader single-writer** communication. The processors are anonymous i.e. have no identities. (We use the term node i or processor P_i as a writing convenience only). A configuration C of the system is the state vector of all processors.

A processor's activity is managed by a **distributed demon** defined in [2, 7]. In each activation the distributed demon activates a subset of the system's processors, all of which execute a single atomic step simultaneously. That is, they read the states of their neighbors, decide whether to change their state and move to their new state. An **execution** of the system is an infinite sequence of configurations $E = c_1, c_2 \cdots$ such that for every i c_{i+1} is a configuration reached from configuration c_i by a single atomic step executed by any subset of processors simultaneously. We say that an execution is **fair** if any node participates in it infinitely often.

A processor can be modeled as a state-machine, having a predetermined set of states. The state transition of a processor is controlled by a **decision function**, f_i, which is a function of its input, its state and the states of its neighbors. The collection of all decision functions is called a **protocol**.

A **uniform protocol** is a protocol in which all the processors are logically equivalent, idenically programmed (i.e. have identical decision functions). Following Dijkstra's observation [6] regarding the mutual exclusion task, we have showen, that solving the network consistency problem, using a uniform protocol, is impossible (see the extended paper). We, therefore, adopt the model of "**almost uniform protocol**" namely, all processors but one are identical and have identical decision functions. We denote the special processor as P_0.

3.2. Self stabilization

Our requirements from a self stabilizing protocol are similar to those in [6]. A self stabilizing protocol should demonstrate legal behavior of the system, namely when starting from any initial configuration (and with any input values) and given enough time, the system should eventually converge to a legal set of configurations for any fair execution. The legality of a configuration depends on the aim of the protocol. Formally, let L be the set of legal configurations. A protocol for the system is **self stabilizing** with respect to L if every infinite fair execution, E, eventually satisfies the following two properties:

1. E enters a configuration c_i that belongs to L.

2. For any $j > i$ and $c_i, c_j \in E$, if $c_i \in L$ then $c_j \in L$ (i.e. once entering L it never leaves it).

In our case a legal configuration is a consistent assignment of values to all the nodes in the network if one exists, and if not, any configuration is legal.

4. A DISTRIBUTED CONSISTENCY-GENERATION PROTOCOL

This section presents a self stabilizing protocol for solving the network consistency problem. It is logically composed of two subprotocols: one simulates the sequential backjumping on DFS (section 4.3), and the other facilitates the desired activation mechanism (section 4.2).

4.1 Neighborhoods and states

We assume the existence of a self-stabilizing algorithm for generating a DFS tree, as a result of which each internal processor, P_i, eventually has one adjacent processor, *parent* (P_i), designated as its parent, and a set of children nodes denoted *children* (P_i). The link leading from *parent* (P_i) to P_i is called **inlink** while the links connecting P_i to its children are called **outlinks**. The rest of P_i's neighbors are divided into two subsets: *ancestors* (P_i), consisting of all neighbors, that reside along the path (in the tree) from the root to P_i, and the set of its successors. For our algorithm a processor can disregard its successors (which are not its children) and observe only the three subsets of neighbors as indicated by figure 3a (for internal nodes) and figure 3c (for leaves). **The root**, having no parent, is played by the special processor P_0 (figure 3b).

We assume that processor P_i (representing variable X_i) has a list of possible values, denoted as $Domain_i$, one of which will be assigned to its state (i.e. to its *value* field in the state register), and a pairwise relation R_{ij} with each neighbor P_j.

The state-register of each processor contains the following fields:

1. A **value** field to which it assigns either one of its domain values or the symbol "*" (to denote a deadend).

2. A **mode** field indicating the processor's "belief" regarding the status of the network. A processor changes the mode from "*on*" to "*off* and vice-versa in accordance with

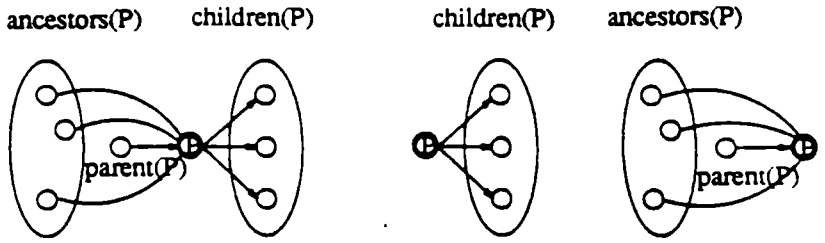


Figure 3: A processor's neighborhood set.

the policy described in section 4.3. The modes of all processors also give an indication whether all the processors have reached a consistent state (all being in an "*off*" mode).

3. Two boolean fields called **parent_tag** and **children_tag**, which are used to control the activity of the processors (section 4.2.)

Additionally, each processor has an ordered domain list which is controlled by a local **domain pointer** (to be explained later), and a local **direction** field indicating whether the algorithm is in its forward or backward phase (to be discussed in section 4.3).

4.2 An activation mechanism

The control protocol is handled by a self-stabilizing activation mechanism. According to this protocol a processor can get a privilege to act, granted to him either by its parent or by its children. A processor is allowed to change its state only if it is privileged.

Our control mechanism is based on a mutual exclusion protocol for two processors called **balance/unbalance**. The balance/unbalance mechanism is a simplified version of Dijkstra's protocol for directed ring [6, 7], and is summarized next.

Consider a system of two processors, P_0 and P_1, each being in one of two states "0" or "1". P_0 changes its state if it equals P_1's state, while P_1 changes its state if it differs from P_0's state. We call a processor that is allowed to change its state **privileged**. In other words, P_0 becomes privileged when the link between the processors is **balanced** (i.e. the states on both its endpoints are identical). It then unbalances the link and P_1 becomes privileged, (the link is unbalanced). P_1 in its turn balances the link. It is easy to see that in every possible configuration there is one and only one privileged processor. Hence this protocol is self stabilizing for the mutual exclusion task and the privilege is passed infinitely often between the two processors. We next extend the balance/unbalance protocol to our needs, assuring, for instance that a node and its ancestor will not be allowed to change their values simultaneously.

Given a DFS spanning tree, every state register contains two fields: **parent_tag**, referring to the inlink and **children_tag**, referring to all the outlinks. A node, *i*, becomes privileged if its inlink is unbalanced and all its outlinks are balanced, namely if the following two conditions are satisfied:

1. for $j = parent(i) : parent\_tag_i \neq children\_tag_j$ (the inlink is unbalanced)

2. $\forall k \in children(i) : children\_tag_i = parent\_tag_k$ [1] (the outlinks are balanced)

A node applies its decision function (described in section 4.3), only when it is privileged (otherwise it leaves its state unchanged), and upon its execution, it passes the privilege accordingly. The privilege can be passed backwards to the parent by balancing the incoming link or forward to the children by unbalancing the outgoing links (i.e. by changing the *parent_tag* or the *children_tag* value accordingly).

We define the **legally-controled configurations**, to be those in which exactly one processor is privileged on every path from the root to a leaf. Figure 4 shows such a configuration. Note how the privilege splits on its way "down". We claim that the control mechanism is self stabilizing, with respect to that legally-controled configurations set [2] (the proof is presented in the extended paper).

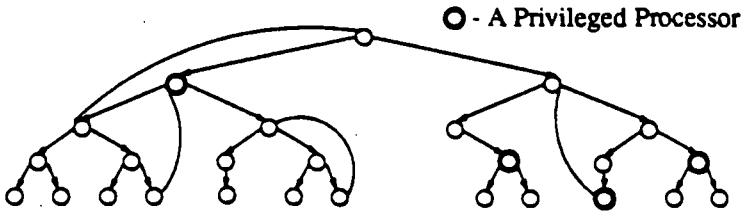


Figure 4: An example for a legal privileges configuration

Once it has become privileged, a processor cannot tell where the privilege came from (i.e. from its parent or from its children). Thus, a processor uses its **direction** field to indicate the source of its privilege. Since during the stable period exactly one processor is privileged on every path from the root to a leaf, the privileges travel along their paths backwards and forwards. The direction field of each processor indicates the direction that the privilege was recently passed by this processor. When passing the privilege to its parent, the processor assigns its direction field the "*backward*" value, while when passing the privilege to its children it assigns the "*forward*" value. Thus, upon receiving the privilege again, it is able to recognize the direction it came from: if *direction* = "*forward*", the privilege was recently passed towards the leaves and therefore it can come only from its children; if *direction* = "*backward*", the privilege was recently passed towards the root and therefore it can come only from its parent. Following are the procedures for privilege passing by P_i.

procedure pass-privilege-to-parent
Begin
1. $parent\_tag_i \leftarrow children\_tag_{parent(i)}$ { balance inlink }
2. $direction_i \leftarrow$ "*backward*"
End.

(1) Note that this is well defined since we can prove that eventually all siblings have the same parent-tag.
(2) We show that our protocol is self-stabilizing with respect to the network consistency task as well as to the legally-controled configurations set (the network converges to $L \cap L'$).

```
procedure pass-privilege-to-children
Begin
  1. for k∈children(i) children_tag_i ← ¬parent_tag_k  { unbalance outlinks }
  2. direction_i ← "forward"
End.
```

4.3 Protocol description

The protocol has a forward and a backward phases, corresponding to the two phases of the sequential algorithm. During the forward phase processors in different subtrees assign consistent values (in parallel) or verify the consistency of their assigned values. When a processor realizes a deadend it assigns its value field a "*" and initiates a backward phase. When the network is consistent (all processors are in an "*off*" mode) the forward and backward phases continue, whereby the forward phase is used to verify the consistency of the network and the backward phase just returns the privilege to the root to start a new forward wave. Once consistency verification is violated the offending processor moves to an "*on*" mode and continues from there.

A processor can be in one of three situations:

1. **Processor P_i is activated by its parent which is in an "*on*" mode** (this is the forward phase of value assignments). In that case some change of value in one of its ancestors might have occurred. It, therefore, resets the domain pointer to point to the beginning of the domain list, finds the first value in its domain that is consistent with all its ancestors, put itself in an "*on*" mode and passes the privilege to its children. If no consistent value exists, it assigns itself the "*" value (a deadend) and passes the privilege to its parent (initiating a backward phase).

2. **Processor P_i is activated by its parent which is in an "*off*" mode.** In that case it verifies the consistency of its current value with its ancestors. If it is consistent it stays in an "*off*" mode and moves privilege to its children. If not, it assigns itself a new value (after resetting the domain pointer to start), moves to an "*on*" mode and passes the privilege to the children.

3. **Processor P_i is activated by its children** (backward phase). If one of the children has a "*" value, the processor selects the next consistent value (after the current pointer) from its domain, resets its domain pointer to point to the assigned value and passes the privilege to the children. If no consistent value is available, it assigns itself a "*" and passes the privilege to its parent (1). If all children have a consistent value, P_i passes the privilege to its parent.

Following we present the algorithms performed by processors P_i, $i \neq 0$, (see figure 5) and the root processor (figure 6).

---

(1) Due to the privilege passing mechanism, when a parent sees one of its children in a deadend it has to wait until all of them have given him the privilege. This is done to guarantee that all subtrees have a consistent view regarding their ancestor's values.

The procedure **compute-next-consistent-value** (figure 5) tests each value which is located after the domain pointer, for consistency. Namely the value is checked against each of *ancestor* (P_i)'s values and the first consistent value is returned. The pointer's location is readjusted accordingly (i.e., to the found value). If no legal value was found the value returned is "*" and the pointer is reset to the beginning of the domain. The procedure **verify-consistency** checks the consistency of current value with ancestors and returns a truth-value (i.e. "*true*" if it is consistent and "*false*" otherwise).

The algorithm performed by the root, P_0, (figure 6) is slightly different and in a way simpler. The root does not check consistency. All it does is assigning a new value at the end of each backward phase, when needed, then initiating a new forward phase.

```
procedure update-state (for any processor except the root)
Begin
 1. read parent (P_i) and children (P_i)
 2. if direction = "backward" then     { privilege came from parent }
 3.    if parent's mode is "on" then
 4.       mode ← "on"
 5.       pointer ← 0
 6.       value ← compute-next-consistent-value
 7.       if value = "*" then
 8.          pass-privilege-to-parent
 9.       else                        { there is a legal consistent value }
10.          pass-privilege-to-children
11.    else                          { parent's mode is "off" }
12.       if verify-consistency = "true" then
13.          mode ← "off"; pass-privilege-to-children
14.       else                        { verify-consistency = "false" }
15.          mode ← "on"
16.          value ← compute-next-consistent-value
17.          if value = "*" then
18.             pass-privilege-to-parent
19.          else                     { there is a legal consistent value }
20.             pass-privilege-to-children
21. else                             { direction = "forward" i.e. privilege came from children }
22.    if ∃k∈ children (P_i) value_k = "*" then
23.       mode ← "on"
24.       value ← compute-next-consistent-value
25.       if value = "*" then        { no consistent value was found }
26.          pass-privilege-to-parent
27.       else                       { there is a legal consistent value }
28.          pass-privilege-to-children
29.    else                          { all children are consistent }
30.       pass-privilege-to-parent
End.
```

Figure 5: The decision function of P_i , $i \neq 0$.

```
procedure root-update-state
Begin
  1. read children (P_0)
  2. if ∃k ∈ children (P_0) value_k = "*" then
  3.     mode ← "on"
  4.     value ← next-value
  5. else                          { all children are consistent }
  6.     mode ← "off"
  7. pass-privilege-to-children
End.
```

Figure 6: The decision function of P_0.

The procedure **next-value** returns the value pointed by the domain pointer and increments the pointer's location (if the end of the domain list is reached, the pointer is reset to the beginning).

In the extended paper we prove the correctness of our protocol. The self-stabilization property of our activation mechanism assures an adequate control for distributedly implemented backtracking. Having this property we can prove the self-stabilization of the "consistency-generation" protocol, namely that eventually the network converges to a legal solution, if one exists, and if not it keeps checking all the possibilities over and over again.

5. COMPLEXITY ANALYSIS

The precise time complexity of the protocol has yet to be formally analyzed. However, a crude estimate can be given of the maximal number of state changes from the time the activation mechanism had stabilized until a final convergence. The worst-case number of states changes depends on the worst-case time of the sequential backjump algorithm. We will present a bound on the search space explored by the sequential algorithm and show that the same bound applies to the number of state changes of our protocol. Our bound improves the one presented in [11].

Let T_m stand for the search space generated by DFS-backjumping when the depth of the DFS tree is m or less. Let b be the maximal brunching degree in the tree and let k bound the domain sizes. Since any assignment of a value to the root node generates b subtrees of depth $m-1$ or less, that can be solved independently, T_m obeys the following recurrence:

$$(1) \quad T_m = k \cdot b \cdot T_{m-1}$$

with $T_0 = k$. Solving this recurrence yields

$$(2) \quad T_m = b^m k^{m+1}$$

(Note that when the tree is balanced we get that $T_m = nk^{m+1}$.)

It is easy to show that the number of state changes of our protocol satisfies exactly the same recurrence. The reason is as follows: Any sequential DFS-backjumping produces a search space smaller or equal to the number of state changes of the distributed protocol. However, there is exactly one run of the sequential algorithm, whose search space is identical to the number of state changes in the protocol, thus the two worst-case are identical.

6. CONCLUSIONS

We have shown that the network consistency problem can be solved distributedly within a connectionist type architecture. The protocol we presented is self-stabilizing, namely its convergence to a consistent solution is guaranteed. The self-stabilizing property renders our model suitable for solving CSPs in dynamic environments. For instance, unexpected changes of some of the domains or some of the constraints will trigger a transient perturbation in the network which will eventually converge to a new solution. Similarly, the protocol can readjust to changes in the network's topology, after generating a new DFS spanning tree. A self stabilizing DFS spanning tree protocol will be presented in the extended paper.

Although we are attacking an NP-complete problem, we have shown that our protocol's complexity is polynomial in networks of bounded DFS depth. Thus the DFS depth can be regarded as a crucial parameter of the rate of convergence in our model. It will be interesting to explore whether the speed of convergence in other models is related to similar parameters.

References

[1] Ballard, D. H., P.C. Gardner, and M. A. Srinivas, "Graph problems and connectionsit architectures," University of Rochester, Rochester, NY, Tech. Rep. 167, 1986.

[2] Burns, J., M. Gouda, and C. L. Wu, "A self-stabilizing token system," in *Proceedings of the 20th Annual Intl. Conf. on System Sciences*, Hawaii: 1987, pp. 218-223.

[3] Dahl, E. D., "Neural networks algorithms for an NP-Complete problem: map and graph coloring," in *Proceedings of the IEEE first Internat. Conf. on Neural Networks*, San Diego: 1987, pp. 113-120.

[4] Dechter, R. and A. Dechter, "Belief maintenance in dynamic constraint networks," in *Proceedings AAAI-88*, St. Paul, Minnesota: 1988.

[5] Dechter, R., "Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition," *Artificial Intelligence*, Vol. 41, No. 3, 1990, pp. 273-312.

[6] Dijkstra, E. W., "Self stabilizing systems in spite of distributed control," in *Communications of the ACM 17,11*, 1974, pp. 643-644.

[7] Dolev, S., A. Israeli, and S. Moran, "Self stabilization of dynamic systems assuming only read/write atomicity," Technion., Tech. Rep. Haifa, Israel, 1990.

[8] Even, S., *Graph Algorithms*, Maryland, USA: Computer Science Press, 1979.

[9] Freuder, E.C., "A Sufficient Condition for Backtrack-Free Search.," *Journal of the ACM*, Vol. 29, No. 1, 1982, pp. 24-32.

[10] Freuder, E. C. and M. J. Quinn, "Parallelism in algorithms that take advantage of stable sets of variables to solve constraint satisfaction problems," University of New Hampshire, Durham, New Hampshire, Tech. Rep. 85-21, 1985.

[11] Freuder, E. C. and M.J. Quinn, "The use of lineal spanning trees to represent constraint satisfactionproblems," University of New Hampshire, Durham, New Hampshire, Tech. Rep. 87-41, 1987.

[12] Gusgen, H.W. and J. Hertzberg, "Some fundamental properties of local constraint propagation," *Artificial Intelligence Journal*, Vol. 36, No. 2, 1988, pp. 237-247.

[13] Haralick, R. M. and G. L. Elliott, "Increasing Tree-search Efficiency for Constraint Satisfaction Problems," *Artificial Intelligence*, Vol. 14, 1980, pp. 263-313.

[14] Mackworth, A. K., "Consistency in Networks of Relations," *Artificial intelligence*, Vol. 8, No. 1, 1977, pp. 99-118.

[15] Rossi, F. and U. Montanari, "Exact solution of networks of constraints using perfect relaxation-," in *Proceedings of the first Intl. Conf. on Principles of Knowledge Representation and Reasoning*, Toronto, Canada: 1989.

[16] Tagliarini, G. A. and E. W. Page , "Solving constraint satisfaction problems with neural networks," in *Proceedings of the IEEE first Internat. Conf. on Neural Networks*, San Diego: 1987, pp. 741-747.

Connectionist Networks for Constraint Satisfaction *

Hans Werner Guesgen
German National Research Center for Computer Science (GMD)
Schloss Birlinghoven, 5205 Sankt Augustin, Fed. Rep. of Germany

Abstract

Algorithms for solving constraint satisfaction problems, i.e. for finding one, several, or all solutions for a set of constraints on a set of variables, have been introduced in a variety of papers in the area of AI. Here, we illustrate how connectionist networks for constraint satisfaction can be implemented.

The idea is to use a connectionist node for each value of each variable and for each tuple of each constraint of the constraint satisfaction problem, and to connect them according to the way in which the constraints are related to the variables. Goedel numbers are used as potentials of the nodes that correspond to variables, representing possible paths of solutions.

1 Introduction

Constraint satisfaction has been applied successfully in many subfields of AI, such as computer vision [22], circuit analysis [20], planning [21], diagnosis [3] [6] [13], and logic programming [10].

A constraint satisfaction problem consists of a constraint network, i.e. a set of variables and a set of constraints on subsets of these variables, and the task of determining one or more tuples of values that satisfy the constraint network. A brute-force approach to finding a solution for a constraint network is to use a backtracking algorithm: values from the domain are tentatively assigned to the variables and the constraints are checked as to whether they are satisfied. If this is not the case, values are backtracked and other values are assigned until a solution is found or no other values can be assigned, i.e. inconsistency is detected.

In general, there is no straightforward (massively) parallel implementation for the backtracking approach. However, there are other algorithms (which are preprocessing methods rather than complete constraint satisfaction algorithms) that can be implemented in parallel. Examples are the arc consistency algorithms in [16], the complexity of which is discussed in [12] and a parallel implementation of which is illustrated in [1]. In particular, it is shown in [1] how connectionist networks can be used to compute arc consistency.

It is the purpose of this paper to introduce connectionist networks that are able to compute global consistency rather than arc consistency. Similar to [1], the nodes used in our networks are in accordance with the unit/value principle (cf. [4]): a separate connectionist node is dedicated to each value of each variable and each tuple of each constraint of the constraint network. This approach is in analogy to the way deKleer represents constraint networks as propositional clauses [14].

The paper is organized as follows: we will first summarize the part of [1] that is essential for our approach and will then show how it can be extended to a complete constraint satisfaction

*Part of this work was performed while the author was at the International Computer Science Institute, 1947 Center Street, Berkeley, California 94704.

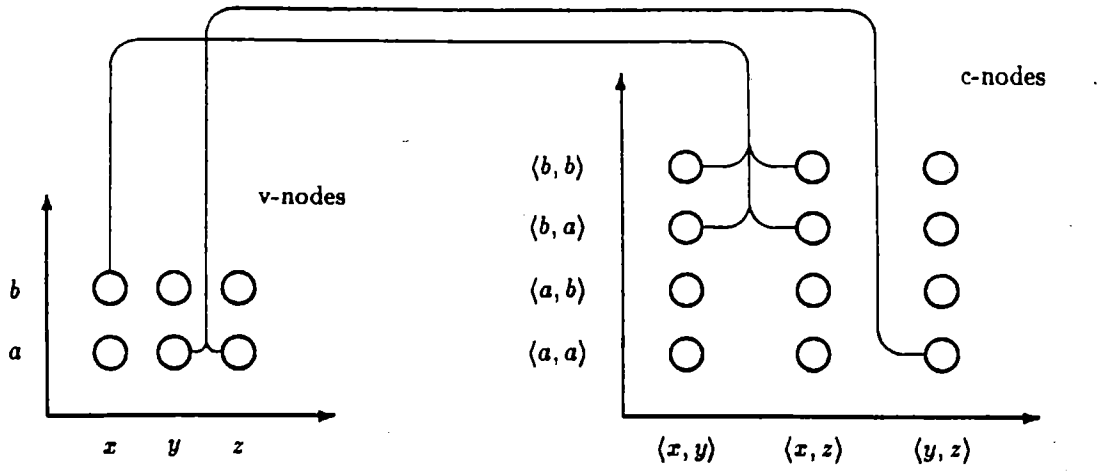


Figure 1: Scheme of a connectionist network for constraint satisfaction.

algorithm, using Goedel numbers to represent possible paths of solutions. Our work is related to the work in symbolic AI that is described in [8], and therefore it is another example of how symbolic algorithms can be implemented in a connectionist manner.

2 Connectionist Representation

Constraints are usually defined on a set of variables V over a domain D. We represent each variable-value pair $\langle x, a\rangle$, $x \in V$, $a \in D$, by a connectionist node (v-node) and denote such a node by $v\langle x, a\rangle$.

A binary constraint on two variables consists of a set of pairs, each pair representing a consistent value assignment for the variables. We introduce a connectionist node (c-node) for each quadruple $\langle x, y, a, b\rangle$ with $x, y \in V$ and $a, b \in D$, and denote such a node by $c\langle x, y, a, b\rangle$. Because of symmetry, $c\langle x, y, a, b\rangle$ and $c\langle y, x, b, a\rangle$ denote the same node.

This representation corresponds directly to the one in [1]. As it is shown there, v-nodes and c-nodes can be connected in such a way that the resulting network computes an arc consistent solution for the corresponding constraint satisfaction problem (cf. figure 1). For that purpose, the nodes are initialized as follows:

- Each v-node obtains potential 1.
- A c-node $c\langle x, y, a, b\rangle$ obtains potential 1, if $\langle a, b\rangle$ is an admissible assignment of values for $\langle x, y\rangle$; else it obtains potential 0.

Since it is more convenient, we will use $v\langle x, a\rangle$, for example, for both: either for referring to the potential of a node or for denoting the node itself.

A v-node is reset to 0 if one cannot find at least one v-node for every other variable such that the constraint between this v-node and the given v-node is satisfied. This rule is called the arc consistency label discarding rule:

$$\text{reset}(v\langle x, a\rangle) = \neg \bigwedge_{y \in V} \bigvee_{b \in D} (v\langle y, b\rangle \wedge c\langle x, y, a, b\rangle)$$

This is equivalent to:

$$v\langle x,a\rangle = \begin{cases} 1 & \text{if } \forall y \in V \exists b \in D : v\langle y,b\rangle \wedge c\langle x,y,a,b\rangle \\ 0 & \text{else} \end{cases}$$

A shortcoming of the label discarding rule is that it computes only arc consistency according to [16]. Although arc consistency may help to find a global solution by restricting the search space, one often needs additional mechanisms to obtain a globally consistent network. We will show in the next section how a connectionist network can be designed to compute global consistency. The approach described in that section may be compared with the one in [15], where signatures are used to maintain variable bindings.

3 Towards Global Consistency

The idea is to apply the same communication scheme as in [1] but to use the potential of a v-node to encode information about how the variable/value pair contributes to a solution (and not only whether or not it does so). The information is composed from the codings that are associated with the c-nodes. In particular, we encode each c-node by a prime number and denote this encoding by a function $e : V^2 \times D^2 \rightarrow P$ from the set of c-nodes to the set of prime numbers.

For example, let $V = \{x, y\}$ and $D = \{a, b\}$, then e may be defined as follows:

$$\begin{array}{ll} e\langle x,x,a,a\rangle = 2 & e\langle x,y,a,a\rangle = e\langle y,x,a,a\rangle = 11 \\ e\langle x,x,b,b\rangle = 3 & e\langle x,y,a,b\rangle = e\langle y,x,b,a\rangle = 13 \\ e\langle y,y,a,a\rangle = 5 & e\langle x,y,b,a\rangle = e\langle y,x,a,b\rangle = 17 \\ e\langle y,y,b,b\rangle = 7 & e\langle x,y,b,b\rangle = e\langle y,x,b,b\rangle = 19 \end{array}$$

Nodes such as $c\langle x,x,a,b\rangle$ with $a \neq b$ do not make sense and therefore are omitted in the coding.

We will again use the same notation for a node and its potential, i.e. the term $c\langle x,y,a,b\rangle$, for example, may denote the connectionist node representing the tuple $\langle a,b\rangle$ of the constraint between x and y, or may denote the potential of that node. It is determined by the context which meaning is intended.

The initial potentials of c-nodes are the same as in [2]. A c-node $c\langle x,y,a,b\rangle$ is assigned the potential 1, if there is a pair $\langle a,b\rangle$ in the constraint between x and y; else it is 0. The initial potential of a v-node $v\langle x,a\rangle$ is determined by the product of the codes of all c-nodes except those that refer to the same variable as the given v-node but to a different value for that variable (i.e. a factor $e\langle x,..,b,..\rangle$ with $b \neq a$ does not occur in the product):

$$\prod_{\substack{y \in V \\ b \in D}} e\langle x,y,a,b\rangle \prod_{\substack{y_1,y_2 \in V \setminus \{x\} \\ b_1,b_2 \in D}} \frac{e\langle y_1,y_2,b_1,b_2\rangle}{2}$$

The factor $\frac{1}{2}$ is due to the fact that $c\langle y_1,y_2,b_1,b_2\rangle$ and $c\langle y_2,y_1,b_2,b_1\rangle$ are identical nodes.

Unlike computing arc consistency (in which a v-node's potential is reset to 0 if it is inconsistent), we will perform here what is called graceful degradation:

1. A c-node receives the potentials of its v-nodes and computes their greatest common divisor (gcd).
2. The gcd is returned to the v-nodes if the c-node has potential 1; else 1 is returned.
3. A v-node computes the least common multiples (lcm) of data coming in from c-nodes that refer to the same variables and combines these by computing their gdc.

The idea is that the potentials of v-nodes shall reflect paths in the network that correspond to solutions of the constraint satisfaction problem. A v-node may be on the same path as another v-node if the c-node between them has potential 1. We start with allowing all paths among the v-nodes. Whenever a part of path is determined that is not admissible, i.e. the corresponding c-node has potential 0, the path is deleted.

This means that global information about solution paths is held locally in the v-nodes of the network. To keep this information consistent, the c-nodes compute the gcd of the potentials of neighboring v-nodes. The gcd reflects that piece of information neighboring v-nodes can agree on. In order to consider alternatives, the v-nodes compute the lcm of data that comes in from c-nodes connecting to the same variable, and combine the results by applying the gcd operator. The alternation between the application of gcd and lcm directly corresponds to the semantics of constraints and their constituting tuples: a constraints network can be viewed as a conjunction of constraints (therefore gcd) wheras a constraint can be viewed as disjunction of tuples (therefore lcm).

More formally, the degradation rule can be denoted as follows:

$$v\langle x,a\rangle \leftarrow \gcd_{y\in V} \operatorname{lcm}_{b\in D} (\operatorname{out}(c\langle x,y,a,b\rangle))$$

with

$$\operatorname{out}(c\langle x,y,a,b\rangle) = \begin{cases} \gcd(v\langle x,a\rangle, v\langle y,b\rangle) & \text{if } c\langle x,y,a,b\rangle = 1 \\ 1 & \text{else} \end{cases}$$

Since the degradation rule is monotonous and discrete, the network finally settles down. After that, the potentials of the v-nodes characterize the set of solutions of the given constraint satisfaction problem. In particular, a solution is given by a subset of v-nodes, W, for which the following holds:

1. Every variable occurs exactly once in W.

2. The potentials of the v-nodes in W are divisible by p, where:

$$p = \prod_{v\langle x,a\rangle, v\langle y,b\rangle \in W} \frac{e\langle x,y,a,b\rangle}{2}$$

(Again, the factor $\frac{1}{2}$ is due to the fact that $c\langle x,y,a,b\rangle$ and $c\langle y,x,b,a\rangle$ are identical nodes.)

We will show in the next section that the approach is sound and complete. We will also provide some upper bound for its space complexity. Before that, however, we will illustrate our approach by a small example. Figure 2 shows a part of a connectionist network for a constraint problem with variables x_1, x_2, x_3, and x_4, which are constrained by binary constraints according to the following table:

Variables	Constraint
x_1, x_2	$\{(a,a),(b,b)\}$
x_2, x_3	$\{(a,a),(b,b),(b,a)\}$
x_3, x_4	$\{(a,a),(b,a)\}$

Here, we use circles for the representation of v-nodes, boxes with solid boundary lines for c-nodes that have potential 1, and boxes with dashed boundary lines for c-nodes that have potential 0. For the sake of simplicity, c-nodes that correspond to universal constraints, i.e. constraints with the whole Cartesian product as relation, have been omitted. This simplification is admissible since all v-nodes are already connected by nonuniversal constraints, guaranteeing that inconsistent codes are removed from the potentials of the v-nodes.

Figure 3 shows a sequence of tables in which listings of v-node potentials and c-node outputs alternate. It illustrates how the network is initialized and how it settles down.

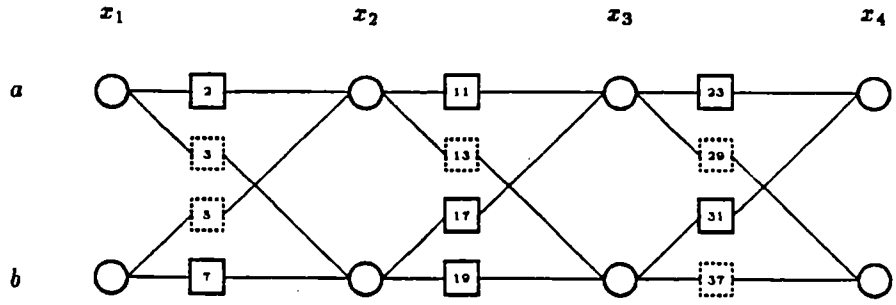


Figure 2: Connectionist network for a simple constraint satisfaction problem.

The example suggests that nodes in the center of the network ($v\langle x_2, a\rangle$, $v\langle x_2, b\rangle$, $v\langle x_3, a\rangle$, and $v\langle x_3, b\rangle$) settle down faster than nodes at the periphery ($v\langle x_1, a\rangle$, $v\langle x_1, b\rangle$, $v\langle x_4, a\rangle$, and $v\langle x_4, b\rangle$). This, however, is only because we simplified the example and left out some connections. In a network with full connectivity, there is no distinction between center nodes and peripheral nodes; therefore, these networks settle down in a more uniform way.

4 Termination, Soundness, and Completeness

The kernel of our approach is the degradation rule as introduced in the previous section, which, on a more abstract level, may be denoted as follows:

$$v\langle x, a\rangle \leftarrow \text{degrade}(\ldots, v\langle x, a\rangle, \ldots)$$

In this context, two observations are important: first, degrade is composed of gcd's and lcm's in such a way that factors are deleted from $v\langle x, a\rangle$ rather than added, and second, the number of factors with which $v\langle x, a\rangle$ is initialized is finite. This implies that the degradation rule terminates, independently of the given constraint satisfaction problem.

We will now show that our approach is sound and complete. For that purpose, we have to answer the following questions:

1. Does every subset W of v-nodes with

   (a) Every variable occurs exactly once in W.

   (b) The potentials of the v-nodes in W are divisible by p, where:

$$p = \prod_{v\langle x,a\rangle, v\langle y,b\rangle \in W} \frac{c\langle x, y, a, b\rangle}{2}$$

   represent a solution, i.e., does the degradation rule always converge to a solution, i.e., is our approach sound?

2. Does a subset W of v-nodes with the above properties exist for each solution of the constraint satisfaction problem, i.e., is our approach complete?

To answer the first question, let us assume that there is a subset of v-nodes, W, for which the above conditions hold but which does not represent a solution of the constraint satisfaction problem. This means that the tuple suggested by W violates at least one constraint of the network. Let $c\langle x, y, a, b\rangle$ be the c-node representing the tuple of the constraint that is violated, then the

V-Node	Initial Potential
$v\langle x_1, a\rangle$	$2 \cdot 3 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_1, b\rangle$	$5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, a\rangle$	$2 \cdot 5 \cdot 11 \cdot 13 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, b\rangle$	$3 \cdot 7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_3, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23 \cdot 29$
$v\langle x_3, b\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 13 \cdot 19 \cdot 31 \cdot 37$
$v\langle x_4, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, b\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 29 \cdot 37$

C-Node	Output
$c\langle x_1, x_2, a, a\rangle$	$2 \cdot 11 \cdot 13 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$c\langle x_1, x_2, a, b\rangle$	1
$c\langle x_1, x_2, b, a\rangle$	1
$c\langle x_1, x_2, b, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$c\langle x_2, x_3, a, a\rangle$	$2 \cdot 5 \cdot 11 \cdot 23 \cdot 29$
$c\langle x_2, x_3, a, b\rangle$	1
$c\langle x_2, x_3, b, a\rangle$	$3 \cdot 7 \cdot 17 \cdot 23 \cdot 29$
$c\langle x_2, x_3, b, b\rangle$	$3 \cdot 7 \cdot 19 \cdot 31 \cdot 37$
$c\langle x_3, x_4, a, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$c\langle x_3, x_4, a, b\rangle$	1
$c\langle x_3, x_4, b, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 13 \cdot 19 \cdot 31$
$c\langle x_3, x_4, b, b\rangle$	1

V-Node	Potential
$v\langle x_1, a\rangle$	$2 \cdot 11 \cdot 13 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_1, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, a\rangle$	$2 \cdot 11 \cdot 23 \cdot 29$
$v\langle x_2, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_3, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$v\langle x_3, b\rangle$	$3 \cdot 7 \cdot 19 \cdot 31$
$v\langle x_4, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 13 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, b\rangle$	1

C-Node	Output
$c\langle x_1, x_2, a, a\rangle$	$2 \cdot 11 \cdot 23 \cdot 29$
$c\langle x_1, x_2, a, b\rangle$	1
$c\langle x_1, x_2, b, a\rangle$	1
$c\langle x_1, x_2, b, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$c\langle x_2, x_3, a, a\rangle$	$2 \cdot 11 \cdot 23$
$c\langle x_2, x_3, a, b\rangle$	1
$c\langle x_2, x_3, b, a\rangle$	$7 \cdot 17 \cdot 23$
$c\langle x_2, x_3, b, b\rangle$	$7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, a, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$c\langle x_3, x_4, a, b\rangle$	1
$c\langle x_3, x_4, b, a\rangle$	$3 \cdot 7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, b, b\rangle$	1

V-Node	Potential
$v\langle x_1, a\rangle$	$2 \cdot 11 \cdot 23 \cdot 29$
$v\langle x_1, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 29 \cdot 31 \cdot 37$
$v\langle x_2, a\rangle$	$2 \cdot 11 \cdot 23$
$v\langle x_2, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_3, a\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$v\langle x_3, b\rangle$	$7 \cdot 19 \cdot 31$
$v\langle x_4, a\rangle$	$2 \cdot 3 \cdot 5 \cdot 7 \cdot 11 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, b\rangle$	1

C-Node	Output
$c\langle x_1, x_2, a, a\rangle$	$2 \cdot 11 \cdot 23$
$c\langle x_1, x_2, a, b\rangle$	1
$c\langle x_1, x_2, b, a\rangle$	1
$c\langle x_1, x_2, b, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$c\langle x_2, x_3, a, a\rangle$	$2 \cdot 11 \cdot 23$
$c\langle x_2, x_3, a, b\rangle$	1
$c\langle x_2, x_3, b, a\rangle$	$7 \cdot 17 \cdot 23$
$c\langle x_2, x_3, b, b\rangle$	$7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, a, a\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$c\langle x_3, x_4, a, b\rangle$	1
$c\langle x_3, x_4, b, a\rangle$	$7 \cdot 19 \cdot 31$
$c\langle x_3, x_4, b, b\rangle$	1

V-Node	Final Potential
$v\langle x_1, a\rangle$	$2 \cdot 11 \cdot 23$
$v\langle x_1, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_2, a\rangle$	$2 \cdot 11 \cdot 23$
$v\langle x_2, b\rangle$	$7 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_3, a\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 23$
$v\langle x_3, b\rangle$	$7 \cdot 19 \cdot 31$
$v\langle x_4, a\rangle$	$2 \cdot 7 \cdot 11 \cdot 17 \cdot 19 \cdot 23 \cdot 31$
$v\langle x_4, b\rangle$	1

Figure 3: Sequence of v-node potentials and c-node outputs.

potential of $c\langle x, y, a, b\rangle$ is 0. Due to the initialization scheme, the potential of a node $v\langle x, a'\rangle$ with $a' \neq a$ does not contain the factor $e\langle x, y, a, b\rangle$; the same holds for $v\langle y, b'\rangle$ with $b' \neq b$. Since the potential of $c\langle x, y, a, b\rangle$ is 0 and since no neighboring node can provide the factor $e\langle x, y, a, b\rangle$, this factor is neither in the potential of $v\langle x, a\rangle$ nor in the potential of $v\langle y, b\rangle$, i.e. these potentials are not divisible by $e\langle x, y, a, b\rangle$. Therefore, they are also not divisible by p (p defined as above). This, however, is in contradiction to the assumption, i.e. every subset of v-nodes for which the above conditions hold represents a solution of the constraint satisfaction problem.

On the other hand, each solution of a given constraint satisfaction problem defines a subset of c-nodes, C, for which the following holds:

1. Each constraint is represented by exactly one c-node.
2. The potentials of the c-nodes are equal to 1.

Let W be the subset of v-nodes that are connected to c-nodes of C, i.e.:

$$W = \{v\langle x, a\rangle \mid \exists y, b : c\langle x, y, a, b\rangle \in C\}$$

It is easy to see that W represents the given solution. Since the potentials of the c-nodes in C are equal to 1, the potentials of the v-nodes in W are divisible by the code of any c-node in C, i.e. they are divisible by p, where p is the product of the codes of c-nodes in C. Thus, W satisfies the conditions proposed above.

With that, we have shown the soundness and completeness of our approach, i.e. the one-to-one relationship between solutions of a given constraint satisfaction problem and special subsets of v-nodes in the corresponding connectionist network. We will now discuss the complexity of our approach.

5 Complexity

Let n be the number of variables of the given constraint satisfaction problem ($n = |V|$), and let m be the number of values in their domain ($m = |D|$). Then, the number of v-nodes and c-nodes can be estimated as follows:

Number of v-nodes:	$O(mn)$
Number of c-nodes:	$O(m^2n^2)$
Total number of nodes:	$O(m^2n^2)$

In addition, we have to consider the local space that is required for storing the Goedel numbers. Each v-node potential is the product of at most $O(m^2n^2)$ factors, corresponding to the c-nodes of the network. These factors could be represented by bit vectors, which facilitates the gcd and lcm operations, reducing them to intersection und union operations. This means that we need additional space of magnitude $O(m^2n^2)$ for each v-node, i.e. $O(m^3n^3)$ additional space in total.

6 Conclusion

We have shown in this paper how a connectionist network can be used to compute the solutions of a constraint satisfaction problem. In particular, we have provided a scheme which allows us to transform an arbitrary binary constraint satisfaction problem, i.e. a set of variables and a set of constraints on these variables, into a connectionist network consisting of v-nodes and c-nodes. C-nodes are initialized with either 1 or 0, depending on whether they correspond to an admissible value combination or an inconsistent one; v-nodes are initialized with a Goedel number, representing the possible paths of solutions.

Unlike former approaches, the connectionist network introduced here computes global consistency rather than arc consistency, i.e. it computes the solutions of a given constraint satisfaction problem. We restricted ourselves to binary constraint satisfaction problems, although our approach can be extended to problems of higher arity in a straightforward manner.

Acknowledgements

Many thanks to Jerry Feldman, Steffen Hoelldobler, and the anonymous referee for their comments on earlier versions of the paper, and to Renee Reynolds for improving the English.

References

[1] P.R. Cooper, M.J. Swain:
Parallelism and Domain Dependence in Constraint Satisfaction.
Technical Report 255, University of Rochester, Computer Science Department, Rochester, New York, 1988.

[2] P.R. Cooper:
Parallel Object Recognition from Structure (The Tinkertoy Project).
Technical Report 301, University of Rochester, Computer Science Department, Rochester, New York, 1989.

[3] R. Davis:
Diagnostic Reasoning Based on Structure and Behavior.
Artificial Intelligence **24** (1984) 347–410.

[4] J.A. Feldman, D.H. Ballard:
Connectionist Models and their Properties.
Cognitive Science **6** (1982) 205–254.

[5] E.C. Freuder:
Synthesizing Constraint Expressions.
Communications of the ACM **21** (1978) 958–966.

[6] H. Geffner, J. Pearl:
An Improved Constraint-Propagation Algorithm for Diagnosis.
In: *Proceedings of the IJCAI-87*, Milan, 1987, 1105–1111.

[7] H.W. Guesgen, J. Hertzberg:
Some Fundamental Properties of Local Constraint Propagation.
Artificial Intelligence **36** (1988) 237–247.

[8] H.W. Guesgen:
CONSAT: A System for Constraint Satisfaction.
Research Notes in Artificial Intelligence, San Mateo: Morgan Kaufmann & London: Pitman, 1989.

[9] H.W. Guesgen:
A Universal Constraint Programming Language.
In: *Proceedings of the IJCAI-89*, Detroit, Michigan, 1989, 60–65.

[10] J. Jaffar, J.L. Lassez:
Constraint Logic Programming.
In: *Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages*, Munich, 1987, 111–119.

[11] S. Kasif:
Parallel Solutions to Constraint Satisfaction Problems.
In: *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, Toronto, Ontario, 1989, 180–188.

[12] S. Kasif:
On the Parallel Complexity of Discrete Relaxation in Constraint Satisfaction Networks.
Artificial Intelligence **45** (1990) 275-286.

[13] J. deKleer, B.C. Williams:
Diagnosing Multiple Faults.
In: *Proceedings of the AAAI-86*, Philadelphia, Pennsylvania, 1986, 132–139.

[14] J. deKleer:
A Comparison of ATMS and CSP Techniques.
In: *Proceedings of the IJCAI-89*, Detroit, Michigan, 1989, 290–296.

[15] T.E. Lange, M.G. Dyer:
High-Level Inferencing in a Connectionist Network.
Technical Report UCLA-AI-89-12, University of California, Los Angeles, California, 1989.

[16] A.K. Mackworth:
Consistency in Networks of Relations.
Artificial Intelligence **8** (1977) 99–118.

[17] R. Mohr, T.C. Henderson:
Arc and Path Consistency Revisited.
Artificial Intelligence **28** (1986) 225–233.

[18] A. Rosenfeld:
Networks of Automata: Some Applications.
IEEE Transactions on Systems, Man, and Cybernetics **5** (1975) 380–383.

[19] A. Samal, T.C. Henderson:
Parallel Consistent Labeling Algorithms.
International Journal of Parallel Programming **16** (1987) 341–364.

[20] R.M. Stallman, G.J. Sussman:
Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis.
Artificial Intelligence **9** (1977) 135–196.

[21] M. Stefik:
Planning with Constraints (MOLGEN: part 1).
Artificial Intelligence **16** (1981) 111–140.

[22] D.L. Waltz:
Generating Semantic Descriptions from Drawings of Scenes with Shadows.
Technical Report AI-TR-271, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1972.

Distributed Constraint Satisfaction for DAI Problems

Makoto Yokoo*, Toru Ishida, and Kazuhiro Kuwabara
NTT Communications and Information Processing Laboratories
1-2356 Take Yokosuka-shi,
Kanagawa 238-03, Japan

Abstract

In this paper, a distributed constraint satisfaction problem (DCSP) is introduced as a general framework for Distributed Artificial Intelligence (DAI) problems. A DCSP is an extension of a constraint satisfaction problem (CSP), and various DAI problems can be formalized as DCSPs.

Various methods for solving DCSPs are developed in this paper. In particular, the newly developed technique called *asynchronous backtracking* allows agents to act asynchronously and concurrently, whereas backtracking in CSP is essentially a sequential procedure. These methods are compared experimentally, and typical DAI problems are mapped into DCSPs.

1 Introduction

Distributed Artificial Intelligence (DAI) is a subfield of AI, which is concerned with how a set of automated agents can work together to solve problems. Recently, [Lesser 90] has presented the idea of viewing DAI as a distributed state space search in order to develop a general framework of DAI. This concept is important because without such general frameworks, it is very difficult to compare alternative approaches (which often make different and quite specific assumptions) or to reproduce results obtained by one approach on slightly different problems. Our goal is to develop a framework for formalizing a subset of DAI problems and methods by extending constraint satisfaction problems (CSPs) [Mackworth 87] to distributed multi-agent environments. In this paper, we define a *distributed constraint satisfaction problem* (DCSP) as a CSP in which multiple agents are involved. CSPs are an important subclass of the problems which can be solved by state space search, and the characteristics of CSPs are well understood both theoretically and experimentally [Haralic and Elliot 80] [Dechter and Meiri 89]. Similarly, DCSPs are a subclass of the problems which can be solved by distributed search. DCSPs are important for the following reasons.

- **Various DAI problems can be formalized as DCSPs.**
  Multi-agent resource allocation problems described in [Conry, *et al.* 88], [Kuwabara and Lesser 89] are examples of a class of problems which can be naturally formalized as DCSPs. By formalizing these problems as DCSPs, the methods for solving DCSPs proposed in this paper can be applied.
- **DCSPs provide a formal framework for studying various DAI methods.**
  There are various options in the methods for solving DCSPs, which influence the efficiency (e.g., the selection order of the values). Agents have to make decisions about these options and these decisions are interrelated. DCSPs serve as a basis for studies such as [Durfee and Lesser 87], in which agents exchange their local plans in order to make agents' decisions coherent. Furthermore, restructuring CSPs in order to solve them more efficiently was studied in [Dechter and Pearl 88], [Fox, *et al.* 89]. Restructuring DCSPs can be considered as equivalent to restructuring the organization of the agents. DCSPs serve as a basis for studies of organization self-design, such as [Ishida, Yokoo, and Gasser 90].

In this paper, we introduce the definition of a DCSP, and describe the various methods for solving DCSPs. In particular, the newly developed technique *asynchronous backtracking* is introduced. Backtracking, which is a standard approach to solve CSPs, is essentially a step-by-step procedure. On the other hand, asynchronous backtracking allows agents to act asynchronously and concurrently. Then, the methods for DCSPs are compared experimentally. Furthermore, we show how typical DAI problems can be mapped into DCSPs and provide appropriate methods for solving them.

2 Distributed Constraint Satisfaction Problem (DCSP)

2.1 CSP

A CSP is defined by m variables $x_1, x_2, ..., x_m$, each of which takes its value from domain $D_1, D_2, ..., D_m$ respectively, and a set of constraints. A constraint is

*Currently staying in Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, Michigan 48109, e-mail: yokoo@caen.engin.umich.edu

defined inclusively by a predicate, i.e., the constraint $P_k(x_{k1}, \ldots, x_{kj})$ is a predicate which is defined on the Cartesian product $D_{k1} \times \ldots \times D_{kj}$. This predicate is true iff the instantiations of these variables are compatible with each other. To find a solution of a CSP is to find an assignment of values to all variables, which satisfies all constraints.

2.2 DCSP

A distributed constraint satisfaction problem (DCSP) is a problem in which variables of a CSP are distributed among agents. Each agent has some variables and tries to instantiate their values. Constraints may exist between variables of different agents, and the instantiations of the variables must satisfy these inter-agent constraints. Formally, there exist n agents $1, 2, \ldots, n$. Each variable x_j belongs to one agent i (this relation is represented as $belongs(x_j, i)$). Constraints are also distributed among agents. The fact that the agent k knows the constraint predicate P_l is represented as $known(P_l, k)$.

We say that a DCSP is solved iff the following conditions are satisfied.

- $\forall i, \forall x_j$ belongs(x_j, i), x_j is instantiated to d_j, and $\forall k, \forall P_l$ known(P_l, k), P_l is true under the assignment $x_1 = d_1, x_2 = d_2, \ldots, x_m = d_m$.

The goal of the methods for solving DCSPs is to reach a solution as quickly as possible with the smallest number of messages. At the same time, considerations for DAI specific problems such as robustness against failures or agent authorities are needed.

In this paper, we make the following assumptions for simplicity. The generalization of these assumptions is discussed in the next section.

- Each agent has exactly one variable.
- Each agent knows all constraint predicates relevant to its variable.

3 Methods for DCSP

The methods for solving CSPs can be divided into two groups, i.e., backtracking algorithms and consistency algorithms [Mackworth 87]. The application of these methods to DCSPs is discussed below.

3.1 Backtracking for DCSP

3.1.1 Synchronous Backtracking

The standard backtracking algorithm for CSP can be simply modified to yield the synchronous backtracking algorithm for DCSP. Assume the instantiation order of the variables is agreed (e.g., from agent 1 to agent n) among agents. Each agent, receiving a partial solution (the instantiations of the preceding variables) from the previous agent, instantiates its variable and attaches the value to the partial solution and sends it to the next agent. If no instantiation to its variable is compatible with the partial solution, the agent sends a *backtracking* message to the previous agent.

This algorithm is weak because agents have to act in predefined sequential order and can not act simultaneously. At each moment, only one agent can act.

3.1.2 Asynchronous Backtracking

The asynchronous backtracking algorithm developed in this paper removes the drawbacks of synchronous backtracking. Each agent acts not in predefined sequential order but concurrently and asynchronously. Each agent instantiates its variable and communicates the variable value to relevant agents. There are two main issues in asynchronous backtracking:

- dealing with asynchronous change in order to avoid irrelevant actions based on obsolete information
- avoiding infinite processing loops

In the following, the main part of the algorithm and the ideas to solve the above issues are described. Then, an example of its application and a proof are shown. The obtained algorithm includes the function of dependency directed-backtracking in CSP [de Kleer 87]. Therefore, it requires less thrashing than synchronous backtracking. For simplicity, we assume every constraint is *binary* (between only two variables).

A constraint satisfaction problem in which all constraints are binary can be represented by a network, where variables are nodes and constraints are links between nodes. Since each agent has exactly one variable, a node also represents an agent. We use the same id for representing an agent and its variable. We assume that every link (a constraint) is *directed*, i.e., for each constraint, one of the two agents is assigned to evaluate the constraint, and the other agent sends its value to the constraint evaluating agent. A link is directed from the value sending agent to the constraint evaluating agent (Figure 1 (a)).

Each agent instantiates its variable concurrently and sends the value to the agents which are connected by outgoing links. After that, agents wait for messages and do actions in response to receiving messages. Figure 2 describes the procedures for receiving two kinds of messages, i.e., an *ok?* message for receiving the value from incoming links (Figure 2 (i)), and a *nogood* message for receiving the request to change its own value from outgoing links (Figure 2 (ii)).

An agent has a set of values from the agents connected by incoming links, which is called an *agent_view*. The fact that x_1's value is 1 is represented by a pair of the agent id and the value, $(x_1, 1)$. Therefore, an agent_view is a set of these pairs, e.g., $\{(x_1, 1), (x_2, 2)\}$. If an *ok?* message is sent from an incoming link, the agent adds the pair to its agent_view and checks whether its own value assignment (represented as (*my_id, my_value*)) is *consistent* with its agent_view. Its own assignment is consistent with the agent_view if all constraints the agent evaluates are true under the value assignments described in the agent_view and (*my_id, my_value*), and all communicated *nogoods* are not *compatible* [1] with the agent_view and (*my_id, my_value*). If its own assignment is not consistent with the agent_view, the agent tries to change *my_value* so that it will be consistent with the agent_view.

[1] A nogood is compatible with the agent_view and (*my_id, my_value*) if all variables in the nogood have the same values in the agent_view and (*my_id, my_value*).

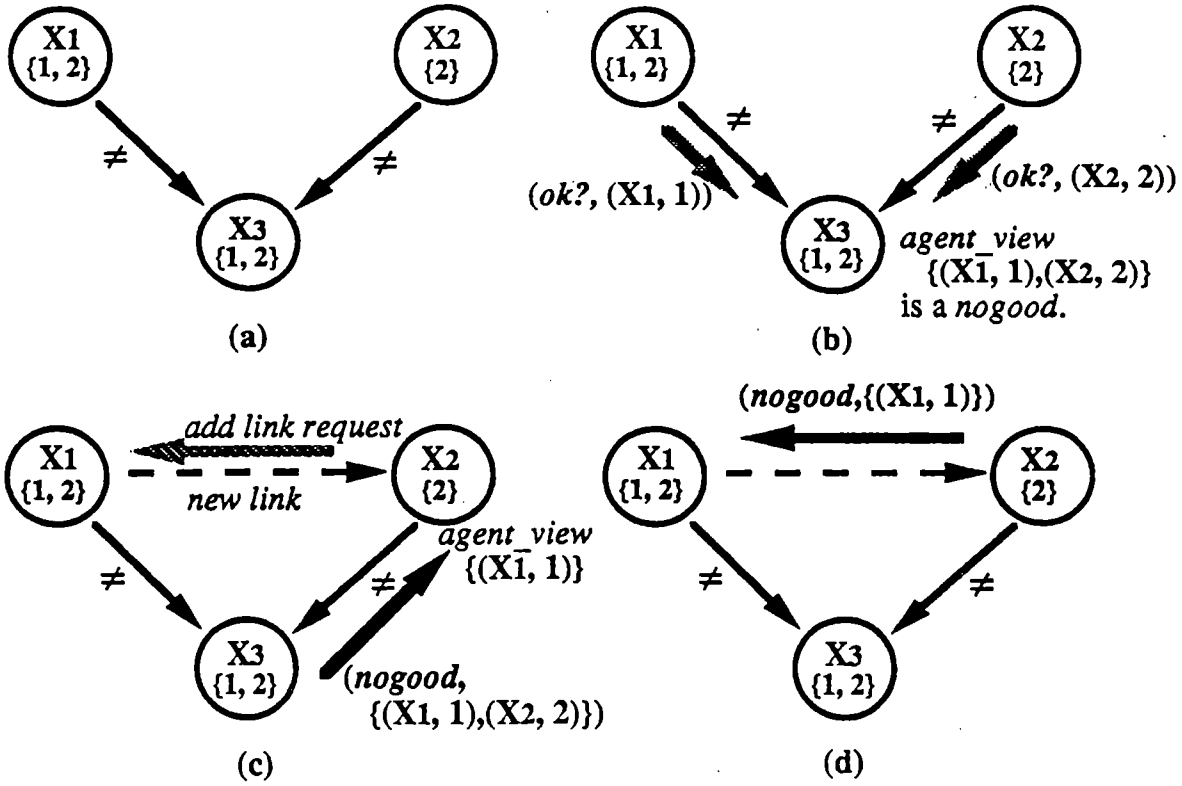


Figure 1: Example of constraint network

A subset of an agent_view is called a *nogood* if the agent is not able to find *my_value* which is consistent with the subset. For example, in Figure 1 (b), there are three agents, x_1, x_2, x_3, with variable domains $\{1,2\}, \{2\}, \{1,2\}$ respectively, and the constraints $x_1 \neq x_3$ and $x_2 \neq x_3$. If agents x_1 and x_2 instantiate their variables to 1 and 2, the agent_view of x_3 will be $\{(x_1,1),(x_2,2)\}$. Since there is no possible value for x_3 which is consistent with this agent_view, this agent_view is a nogood. If an agent finds a subset of its agent_view is a nogood, the assignments of other agents must be changed. Therefore, the agent causes a *backtrack* (Figure 2 (iii)) and sends a *nogood* message to one of the other agents.

Dealing with asynchronous changes. One difficulty is caused by the fact that an agent_view is subject to incessant changes, since agents change their instantiations asynchronously. Coincident with sending a nogood message, the value of an agent's variable which is relevant to the failure may be changed. After that change, there would be no need for backtracking.

The idea for solving this difficulty is *context attachment*, i.e., each message is coupled with the nogood, which is the context of backtracking. After receiving this message, the receiver checks whether the nogood is *compatible* with its agent_view and its own assignment, and changes its value only if the nogood is compatible (Figure 2 (ii-a)). Since the nogood attached to a *nogood* message indicates the cause of the failure, asynchronous backtracking includes the function of dependency directed-backtracking in CSPs.

A nogood can be viewed as a new constraint obtained from original constraints. For example, in Figure 1 (c), the nogood $\{(x_1,1),(x_2,2)\}$ represents a constraint between x_1 and x_2. There is no link between x_1 and x_2 originally. In order to evaluate this new constraint, a new link must be added between x_1 and x_2. Therefore, after receiving the *nogood* message, agent x_2 asks x_1 to add a link between them. In general, even if all original constraints are binary, newly obtained constraints can be among more than 2 variables. In such a case, one of the agents in the constraint will be an evaluator and links will be added between each of non-evaluator agents and the evaluator.

Avoiding infinite processing loops. If agents change their values again and again and never reach a stable state, they are in an infinite processing loop. An infinite processing loop can occur if there exists a value changing loop of agents, e.g., a change in x_1 causes x_2 to change, then this change in x_2 causes x_3 to change, and then this change causes x_1 to change, and so on. In the network representation, such a loop is represented by a cycle of directed links in the network.

One way to avoid cycles in a network is to use a total order relationship among nodes. If each node has an unique id, and a link is directed from the smaller node to the larger node, there will be no cycle in the network. This means that each agent has an unique id, and for each constraint, the larger agent will be an evaluator, and the smaller agent will send an *ok?* message to the evaluator. Furthermore, if a nogood is found, a *nogood* message is sent to the largest agent in

```
when received (ok?, (sendor_id,value)) do — (i)
  add (sendor_id,value) to agent_view;
  when my_value and agent_view are
    inconsistent do
      change my_value to a new consistent value;
      when can not find such a value do backtrack;
        change my_value to a new consistent value;
      end do;
      send (ok?, (my_id,my_value)) to its outgoing links;
  end do;
end do;

when received (nogood, sendor_id, nogood) do — (ii)
  record nogood;
  when (id,value) where id is not connected
        is contained in nogood do
    request id to add a link from id to my_id
      and add (id,value) to agent_view;
  end do;
  if agent_view and my_value are
   incompatible with nogood — (ii-a)
    then send (ok?, (my_id,my_value)) to sendor_id;
    else change my_value to a new consistent value;
      when can not find such a value do backtrack;
        change my_value to a new consistent value;
      end do;
    send (ok?, (my_id,my_value)) to its outgoing links;
  end if;

procedure backtrack — (iii)
begin
  nogoods ←
      {V_s | V_s=inconsistent subset of agent_view};
  when {}∈ nogoods do
    broadcast to other agents that there is
      no solution, terminate this algorithm;
  end do;
  for each V_s = {(id_1, v_1), ...} ∈ nogoods do;
    select (id_j, v_j)
    where id_j is the largest in V_s; — (iii-a)
    send (nogood, my_id, V_s) to id_j;
    remove (id_j, v_j) from agent_view;
  end do;
end backtrack;
```

Figure 2: Procedure for receiving messages

the nogood (Figure 2 (iii-a)), and the largest agent will be an evaluator and links are added between each of non-evaluator agents in the nogood and the evaluator. Similar techniques to this unique id method are used for avoiding deadlock in distributed database systems [Rosenkrantz, *et al.* 78].

The required knowledge of each agent for this unique id method is much more local than that for synchronous backtracking. In synchronous backtracking, agents must act in predefined sequential order. Such sequential order can not be obtained easily just by giving an unique id to each agent. Each agent must know who are the previous and next agent in synchronous backtracking. On the other hand, in the unique id method for asynchronous backtracking, each agent has to know only the relations between their neighbors, i.e., which is the larger. This knowledge for asynchronous backtracking is much more local than that for synchronous backtracking. For example, all people in the world can be totally ordered by their heights, and it is easy for a person to tell the height relation (which is the taller) between his neighbors. However, it will be very difficult (almost impossible) for him to determine the person whose height is next to him in the world.

As for CSPs, the order of the variables greatly affects the search efficiency. Further research is needed in order to introduce variable ordering heuristics into asynchronous backtracking.

An example. In Figure 1 (b), by receiving *ok?* messages from x_1 and x_2, the agent_view of x_3 will be $\{(x_1, 1), (x_2, 2)\}$. Since there is no possible value for x_3 which is consistent with this agent_view, this agent_view is a nogood. Agent x_3 chooses the largest agent in the agent_view, agent x_2, and sends a *nogood* message with the nogood, and removes $(x_2, 2)$ from the agent_view. By receiving this *nogood* message, agent x_2 records this nogood. This nogood, $\{(x_1, 1), (x_2, 2)\}$ contains agent x_1, which is not connected with x_2 by a link. Therefore, a new link must be added between x_1 and x_2. Agent x_2 requests x_1 to send x_1's value to x_2, and adds $(x_1, 1)$ to its agent_view (Figure 1 (c)). Agent x_2 checks whether its value is consistent with the agent_view. Since the nogood received from agent x_3 is compatible with its assignment $(x_2, 2)$ and its agent_view $\{(x_1, 1)\}$, the assignment $(x_2, 2)$ is inconsistent with the agent_view. The agent_view $\{(x_1, 1)\}$ is a nogood because x_2 has no other possible values. There is only one agent in this nogood, i.e., agent x_1, so agent x_2 sends a *nogood* message to agent x_1 (Figure 1 (d)).

Proof. If there exists a solution, this algorithm reaches a stable state where all variable values satisfy all constraints, and all agents are waiting for an incoming message[2]. If there exists no solution, this algorithm finds the fact that there is no solution and terminates.

[2]It must be mentioned that the way to determine that agents as a whole reach a stable state is not contained in this algorithm. In order to detect the stable state, distributed termination detection algorithms such as [Chandy and Lamport 85] are needed.

The soundness of this algorithm, i.e., if the agents reach a stable state, then all variable values satisfy all constraints, is obvious because agents are never in a stable state unless their constraints are satisfied. We show that this algorithm is complete, i.e., the algorithm finds a solution if there exists one, and if there is no solution, the algorithm finds the fact that there exists no solution.

This algorithm terminates if and only if an empty set is found to be a nogood. Logically, a nogood represents a set of assignments from which a contradiction is derived. If a contradiction is derived from an empty set, it means a contradiction can be derived from any assignment, and no assignment can be a solution.

We show the proof that there will be no infinite processing loop. If an infinite processing loop exists, the agent x_1, which has the smallest id, is either in a stable state (case-1) or in an infinite processing loop (case-2).

In case-1, assume that agents x_1 to x_{k-1} ($k > 2$) are in a stable state, and agent x_k is in an infinite processing loop. Since agents x_1 to x_{k-1} ($k > 2$) are in a stable state, we can conclude that the agent_views of x_k to x_n will contain the correct values of x_1 to x_{k-1}. In this case, the only messages agent x_k receives are *nogood* messages from agents whose ids are larger than k. Since agents x_1 to x_{k-1} are in a stable state, the nogoods agent x_k receives is compatible with the agent_view of x_k, and agent x_k changes its value by receiving these *nogood* messages. Agent x_k must send a *nogood* message to one of the agents whose id is smaller than k sooner or later, because the domain of its variable is finite. This fact contradicts the assumption that agents x_1 to x_{k-1} are in a stable state. In case-2, the only messages agent x_1 receives are *nogood* messages. If agent x_1 receives *nogood* messages for all possible values, an empty set will be a nogood and this algorithm terminates. If this algorithm does not terminate, agent x_1 must reach a stable state sooner or later, and this fact contradicts the assumption that agent x_1 is in an infinite processing loop. Since both case-1 and case-2 are impossible, there will be no infinite processing loop.

By using this fact, we show how agents will reach a solution. Agent x_1 will reach a stable state where the value of x_1 is a part of a solution if there exists a solution. If the choice of x_1 is a part of a solution, agent x_1 never change the value since no valid *nogood* message will be received, and eventually the agent_views of agent 2 to agent n will contain the correct value of x_1. If the choice of x_1 is not a part of a solution, since agent x_2 to agent x_n will not fall into an infinite processing loop and can not reach a stable state, x_1 will receive a *nogood* message from one of x_2 to x_n and change its value. Similarly, it can be shown that if agent x_1 to x_{k-1} are in a stable state where the values of x_1 to x_{k-1} are a part of a solution, agent x_k will also reach a stable state where the values of x_1 to x_k are a part of a solution. Therefore, we can show inductively that all agents will reach a solution. If there exists no solution, since there will be no infinite loop and agents can not reach a stable state, this algorithm will terminate by finding an empty nogood.

3.2 Consistency Algorithm for DCSP

For DCSPs, consistency algorithms [Mackworth 87] can be used as a preprocessing procedure before backtracking. Applying various consistency-algorithms developed for CSPs to DCSPs is not so straightforward, although they are generally assumed to be a collection of local procedures for variables. Actually, 2-consistency algorithms are a collection of local procedures for variables which are connected with each other, and easy to apply to DCSPs. However, most of k-consistency algorithms [Fruder 78] require to achieve $1, 2, \ldots, k$ consistency sequentially. Therefore, applying these algorithms to DCSPs requires global synchronization, i.e., all agents must make sure that l-1-consistency is achieved globally before achieving l-consistency.

On the other hand, the k-consistency algorithm in the ATMS framework [de Kleer 89] is essentially monotonic and the final result is not affected by the order of the local procedures. Therefore, the ATMS-based k-consistency algorithm is suitable for DCSPs.

The ATMS-based k-consistency algorithm for DCSPs is described as follows.

1. Exchange the domains of the variables between interrelated agents.
2. Generate nogoods using constraints, then generate new nogoods whose lengths are less than k using hyper-resolution rules, send the new nogoods to relevant agents.
3. Generate new nogoods from communicated nogoods using hyper-resolution rules, repeat until no new nogood can be obtained.

This algorithm can be modified by restricting the application of hyper-resolution rules so that the ids of agents in newly generated nogoods must be smaller than the generator id. The obtained result is equivalent to the directed k-consistency in [Dechter and Pearl 88].

3.3 Generalization of Methods for DCSPs

We assume that one agent has exactly one variable and the agent knows all constraint predicates which are relevant to its variable. The first assumption can be relaxed so that an agent may have several variables, since an agent which has several variables can act as if each of its variables belongs to distinct agents. The second assumption can be relaxed so that an agent do not have to know all constraints relevant to its variables, but has to know the ids of agents which know the relevant constraints. In that case, agents send their values to the evaluators, i.e., the agents which have relevant constraints.

4 Comparison of Methods for DCSP

In this section, the various methods for DCSP described in the previous section are compared. The efficiency of these methods is measured by two simulator-based values, i.e., *number of cycles* and *number of messages*. One cycle consists of the time that an agent reads all messages in its input buffer, does local processing, and sends all messages to other agents. By assuming all agents

act synchronously (all of them enter the first cycle at the same time, and then enter the second cycle at the same time, and so on), the number of cycles which is required to finish the algorithm, and the total number of messages are measured. The number of messages corresponds to the *message complexity*, and the number of cycles corresponds to the *ideal time*, which are widely used for evaluating distributed algorithms.

4.1 Tradeoff between asynchronous and synchronous backtracking

The following results are obtained by experiments on many example problems.

- The asynchronous backtracking algorithm includes the function of dependency-directed backtracking (DDB), and each agent acts concurrently. Therefore, the required cycles to reach a solution for asynchronous backtracking are fewer than for synchronous backtracking, or even for synchronous backtracking which incorporates DDB[3].
- On the other hand, asynchronous backtracking, in which the agents connected by constraints communicate with each other concurrently, usually requires more messages than synchronous backtracking, in which only one message is required for one cycle. Therefore, there exists a tradeoff between the number of cycles and the number of messages.
- As the number of constraints increases, the number of messages for asynchronous backtracking increases more rapidly than for synchronous backtracking. Therefore, asynchronous backtracking is suitable for problems which have natural locality, i.e., interaction between agents is relatively small.
- If the problem has locality, as the number of agents increases, the number of cycles for asynchronous backtracking increases more slowly than for synchronous backtracking. Therefore, asynchronous backtracking has an advantage to synchronous backtracking in large problems, as long as the problems have locality.

Figure 3 shows results of nqueens problems. There exist n agents, each of which tries to position its queen so that the queens do not threaten each other. Obviously, the number of cycles for asynchronous backtracking is much less than for synchronous backtracking. On the other hand, as the number of queens increases, the number of messages for asynchronous backtracking increases more rapidly than for synchronous backtracking. This is because in the nqueens problem, all agents are connected with each other by constraints. We can see that the increase in messages is approximately linear to the number of constraints, which is quadratic in n (the number of queens).

Figure 4 shows the results of randomly generated problems, as a function of the number of constraints. These problems are generated given the following parameters: the number of agents n, the number of values

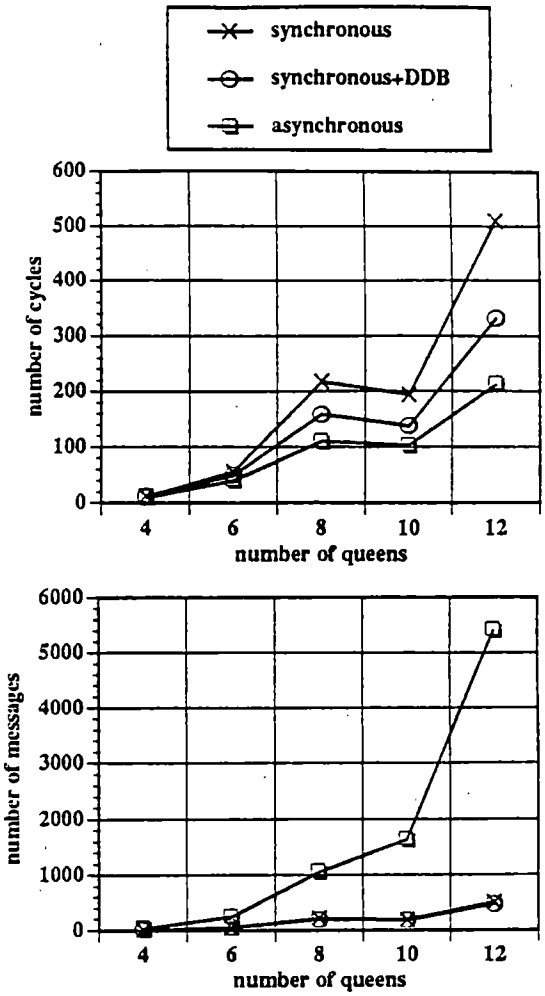


Figure 3: Experimental results for nqueens problems

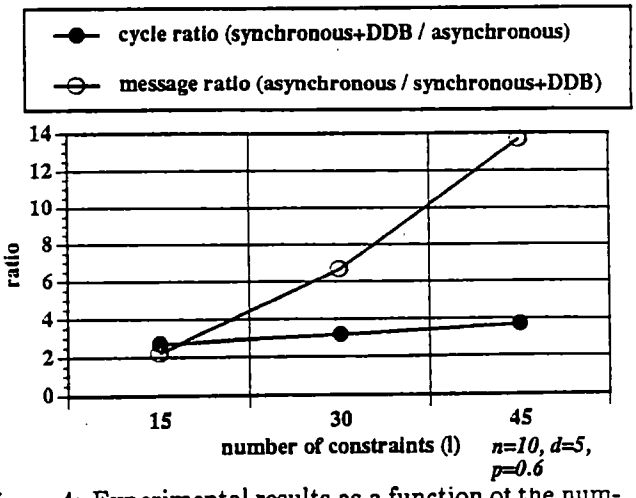


Figure 4: Experimental results as a function of the number of constraints

[3]Backtracking is done to the nearest agent which is relevant to the failure and agents keep track of the failure.

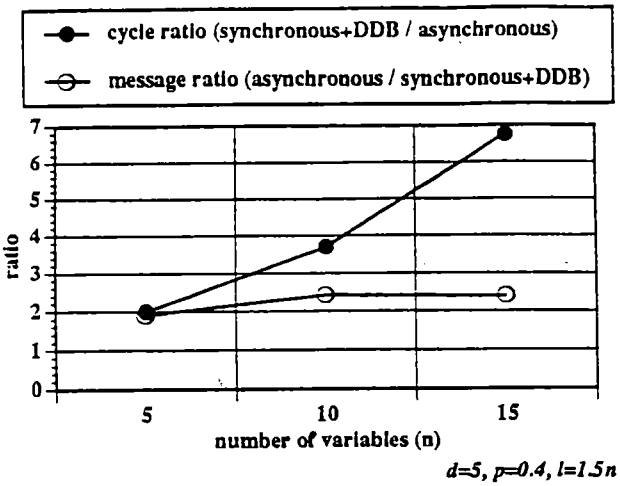


Figure 5: Experimental results as a function of the number of variables

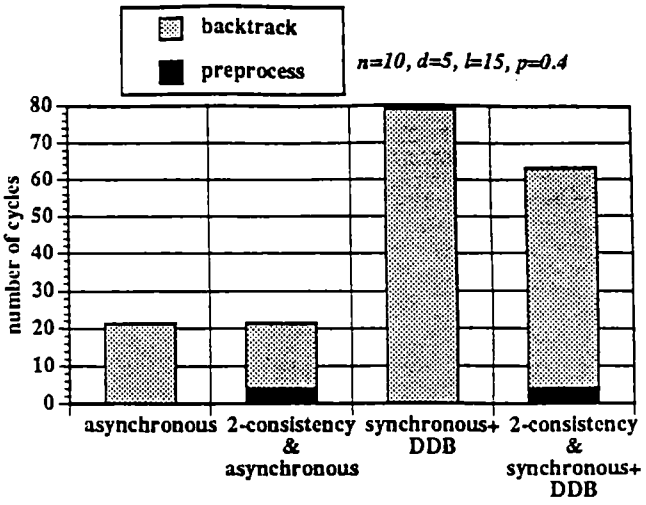


Figure 6: Effect of 2-consistency algorithm for synchronous and asynchronous backtracking

of one variable d (all variables have the same number of values), the number of constraints l, the probability that a value pair of two variables, which are connected by a constraint, satisfies the constraint. Figure 4 shows the results as a function of l (the number of constraints), given n, d, p. For each set of parameters, 25 problems are generated and the graph shows the ratio of the average of these problems. For the number of cycles, the graph shows the ratio of synchronous/asynchronous, and for the number of messages, the ratio of asynchronous/synchronous. We can see that as the number of constraints increases, the ratio of the messages increases. Similar results are obtained by varying n, d, p. In most of DAI problems, we can expect that there exists natural locality. For example, in the example problem in [Conry, *et al.* 88] (routing problem of communication network), the number of agents is $n = 5$, and the number of constraints is $l = 6$.

Figure 5 shows the results as a function of n, given d, p, and l is set to $1.5 \times n$. As long as the average number of the constraints for one agent is fixed, the number of cycles for asynchronous backtracking increases more slowly than for synchronous backtracking. Asynchronous backtracking has an advantage to synchronous backtracking for large problems which have locality, since agents can exploit more concurrency.

4.2 Tradeoff between consistency algorithms and backtracking

Finding an appropriate combination of consistency algorithms and backtracking is an important issue in DCSP. If the problems have a small number of solutions (typically only one solution), 2 or 3-consistency algorithms are effective. On the other hand, if the problems have several solutions, applying backtracking without any preprocessing can be more effective. For example, in the 8queens problem, 2 or 3-consistency algorithms are totally useless (no new nogood is generated), and more powerful consistency algorithms are inefficient because too many nogoods are generated. On the other hand, in line drawing recognition problems, the 2-consistency algorithm described in this paper eliminates all futile backtracking, and the routing problems in communication networks described in [Conry, *et al.* 88], [Kuwabara and Lesser 89] (no solution exists because constraints are too strong), the fact that there is no solution is shown by achieving 2 or 3-consistency. These results are similar to those of CSPs.

By the experiments on many randomly generated problems, the following results are obtained.

- The cost of preprocessing and the effect of preprocessing is almost equal for the 2-consistency algorithm.
- By achieving 3-consistency, the total number of cycles can be reduced, but the total number of messages increases.

[Dechter and Meiri 89] shows that in CSP, 2-consistency algorithm is usually effective for reducing thrashing, and n-consistency algorithms where $n > 2$ require too much preprocessing costs. However, in DCSP, 2-consistency algorithm is not effective for reducing required cycles. Figure 6 shows the total number of cycles for asynchronous backtracking after achieving 2-consistency and synchronous backtracking with DDB after achieving 2-consistency for given n, d, p, l.

In synchronous backtracking, the variable values are determined sequentially. If there exists a strongly constrained variable, it is very effective to propagate the constraints from that variable by preprocessing in order to reduce useless backtracking (which occurs before determining the value of the strongly constrained variable). On the other hand, all variable values are determined asynchronously in asynchronous backtracking and the constraints are propagated immediately from the strongly constrained variable. Therefore, the 2-consistency algorithm is not as effective for asynchronous backtracking as for synchronous backtracking.

5 DCSP in DAI

In this section, we show how allocation problems and recognition problems, which are typical in DAI, can be

mapped into DCSPs.

5.1 DCSP in Allocation Problems

If the problem is to allocate tasks or resources to agents and there exist inter-agent constraints, such a problem can be formalized as a DCSP by viewing each task or resource as a variable and the possible assignments as values. Examples are discussed below.

Contract net.
The contract net protocol [Smith 80] is a protocol for assigning tasks to agents. Since the contracts made during task allocation are assumed to be independent, the aspect of the task allocation as a CSP, i.e., allocating tasks so that constraints are satisfied, is not emphasized in the contract net protocol. However, when the number of tasks increases, it will be an important issue to allocate tasks so that as many tasks can be executed simultaneously as possible.

Multistage negotiation.
The multistage negotiation [Conry, *et al.* 88] deals with the case in which tasks are not independent and there are several ways to perform a task (plans). The goal of multistage negotiation is to find the combination of plans which enables all tasks to be executed simultaneously. [Conry, *et al.* 89] and [Kuwabara and Lesser 89] formalized a class of allocation problems using goals and plans, and described algorithms for finding conflicting goals. This class of problems is a subset of DCSPs and the essential part of these algorithms is equivalent to the hyper-resolution rules. These algorithms try to achieve n-consistency.

For the routing problems in communication networks described in [Kuwabara and Lesser 89] (no solution exists since the problems are over-constrained), 2 or 3-consistency algorithm presented in this paper is sufficient to find that no solution exists. On the other hand, for similar routing problems which have multiple possible solutions, k-consistency algorithms for $k \leq 3$ are not effective for reducing futile backtracking, and k-consistency algorithms for $k \geq 4$ are inefficient since they produce too many nogoods. Therefore, using backtracking without preprocessing is more efficient.

5.2 DCSP in Recognition Problems

A recognition problem can be viewed as a problem to find a compatible set of hypotheses, which corresponds to the possible interpretations of input data. A recognition problem can be mapped into a CSP by viewing possible interpretations as possible variable values. Several examples are described below.

Waltz's labeling problem of line drawings.
Waltz's labeling problem [Waltz 75] is a recognition problem of line drawings. The problem to find the compatible interpretations of agents, each of which is assigned a different part of a line drawing, can be formalized as a DCSP. For the DCSP version of the line drawing recognition problem in [Winston 77], the algorithm for achieving 2-consistency (arc-consistency) described in this paper eliminates all futile backtracking.

Seismic Interpretation Problem.
[Mason and Johnson 89]
proposed the Distributed ATMS as a framework for solving seismic interpretation problems, in which each agent finds the compatible combination of interpretations of its input sensor data. The ATMS supports assumption-based reasoning of each agent. If the interpretations of different agents have to be compatible with each other, consistency algorithms can be used for eliminating hopeless interpretations. In the application problem of [Mason and Johnson 89], however, the interpretations of different agents are possibly incompatible if the agents have different opinions and can not agree with each other.

DVMT.
DVMT [Lesser and Corkill 83] utilizes a method called FA/C, where each agent solves a sub-problem and eliminates possibilities by exchanging the intermediate results (result sharing), and finally reaches a mutually consistent solution. The method for solving a DCSP, where each agent first finds possible solutions for its sub-problem, and exchanges these solutions and eliminates the possibilities by consistency algorithms can be regarded as a kind of FA/C. However, further research is needed for formalizing the high-level coordination framework in DVMT, where only abstract information is exchanged.

5.3 Other Related Problem

Distributed Truth Maintenance.
[Bridgeland and Huhns 90] presented a distributed truth maintenance algorithm. The truth maintenance tasks are essentially solving a DCSP where each variable can be either IN or OUT. However, truth maintenance tasks are incremental, i.e., a new justification (new constraint) will be added to a stable state (where all constraints are satisfied), and some variable values must be changed. In [Bridgeland and Huhns 90], instead of re-solving the whole problem, some variables are selected first, and a DCSP among selected variables is solved. If the DCSP can not be solved, then more variables are added and a DCSP among these variables is solved, and so on. The algorithm used for solving DCSPs in [Bridgeland and Huhns 90] is a kind of synchronous backtracking. Therefore, the distributed truth maintenance algorithm can be optimized by introducing asynchronous backtracking and consistency algorithms.

6 Conclusions

The distributed constraint satisfaction problem was formalized and various methods for solving DCSPs and comparisons of these methods were presented. Furthermore, the formalization of typical DAI problems as DCSPs and appropriate methods for solving these problems were described.

The future issues are as follows.

- **Introducing heuristics**
  Various heuristics proposed in CSP can be introduced into DCSP. Especially, heuristic repair method [Minton, *et al.* 90] is very powerful and

easy to introduce into DCSP. In [Minton, *et al.* 90], variable values are changed so that the number of constraint violations will be minimized. Introducing this heuristic into DCSP can be considered as equivalent to making agents cooperative.

- **Extending DCSPs**
  Using the analogous techniques described in [Fruder 90], DCSPs can be extended so that the number of variable values can be increased dynamically. By this extension, DCSPs can formalize the problems in which the local solutions of each agent are obtained incrementally.
- **Studying various DAI methods using the DCSP framework**
  We are now trying to formalize various DAI methods (e.g., methods for achieving coherent behaviors, methods for organization self design) using the DCSP framework. Our goal is to obtain quantitative comparison results of alternative approaches.

References

[Bridgeland and Huhns 90] Bridgeland, D.M. and Huhns, M.N.: Distributed Truth Maintenance, *Proc. of AAAI-90*, pp.72–77, 1990.

[Chandy and Lamport 85] Chandy, K.M. and Lamport, L : Distributed Snapshots: Determining Global States of Distributed Systems, *ACM Trans. on Computer Systems*, Vol.3, No.1, pp.63–75, 1985

[Conry, *et al.* 88] Conry, S.E., Meyer, R.E. and Lesser, V.R. : Multistage Negotiation in Distributed Planning, In Alan H.Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, Morgan Kaufmann, pp. 367–384,1988.

[Conry, *et al.* 89] Conry, S.E., Meyer, R.E. and Pope, R.P. : Mechanisms for Assessing Nonlocal Impact of Local Decisions in Distributed Planning, In Les Gasser and M.N. Huhns, editors, *Distributed Artificial Intelligence Vol.II*, Morgan Kaufmann, pp. 245–258, 1989.

[Dechter and Meiri 89] Dechter, R. and Meiri, I.: Experimental Evaluation of Preprocessing Techniques in Constraint Satisfaction Problems, *Proc. of IJCAI-89*, pp.271–277, 1989.

[Dechter and Pearl 88] Dechter, R. and Pearl, J. : Tree-clustering Schemes for Constraint-Processing, *Proc. of AAAI-88*, pp.150–154,1988.

[de Kleer 87] de Kleer, J : Dependency-directed Backtracking, In S.C.Shapiro editor, *Encyclopedia of Artificial Intelligence*, John Wiley & Sons, pp.47–48, 1987.

[de Kleer 89] de Kleer, J : A Comparison of ATMS and CSP Techniques, *Proc. of IJCAI-89*, pp.290–296, 1989.

[Durfee and Lesser 87] Durfee, E.H. and Lesser, V.R. : Using Partial Global Plans to Coordinate Distributed Problem Solvers, *Proc. of IJCAI-87*, pp.875–883, 1987.

[Fox, *et al.* 89] Fox, M.S., Sadeh,N. and Baykan, C. : Constrained Heuristic Search *Proc. of IJCAI-89* , pp.309–315, 1989.

[Fruder 78] Fruder, E.C. : Synthesizing Constraint Expressions, *Communications of the ACM* 21(11), pp.958–966, 1978.

[Fruder 90] Fruder, E.C. : Partial Constraint Satisfaction, *Proc. of IJCAI-89* , pp.278–283, 1989.

[Haralic and Elliot 80] Haralic, R.M. and Elliot, G.L., Increasing Tree Search Efficiency for Constraint Satisfaction Problems, *Artificial Intelligence*, vol.14, pp. 263–313, 1980.

[Ishida, Yokoo, and Gasser 90] Ishida, T., Yokoo, M., and Gasser, L : An Organizational Approach to Adaptive Production Systems, *Proc. of AAAI-90*, pp.52–58, 1990.

[Kuwabara and Lesser 89] Kuwabara, K. and Lesser, V.R. : Extended Protocol for Multistage Negotiation, In M. Benda editor, *Proc. of 9th Workshop on Distributed Artificial Intelligence*, pp. 129–161, 1989.

[Lesser and Corkill 83] Lesser, V.R. and Corkill, D.D. : The Distributed Vehicle Monitoring Testbed: A Tool for Investigating Distributed Problem Solving Networks, *AI Magazine*, Fall, pp. 15–33, 1983.

[Lesser 90] Lesser, V.R. : An Overview of DAI: Viewing Distributed AI as Distributed Search, *Journal of Japanese Society for Artificial Intelligence*, Vol.5, No.4, 1990.

[Mackworth 87] Mackworth, A.K. : Constraint Satisfaction, In S.C.Shapiro, editor, *Encyclopedia of Artificial Intelligence*, John Wiley & Sons, pp.205–211, 1987.

[Mason and Johnson 89] Mason, C.L. and Johnson, R.R. : DATMS: A Framework for Distributed Assumption Based Reasoning, In Les Gasser and M.N. Huhns, editors, *Distributed Artificial Intelligence Vol.II*, pp. 293–318, Morgan Kaufmann, 1989.

[Minton, *et al.* 90] Minton, S., Johnston, M.D., Philips, A.B., and Laird, P.: Solving Large-Scale Constraint Satisfaction and Scheduling Problems Using a Heuristic Repair Method, *Proc. of AAAI-90*, pp.17–24, 1990.

[Rosenkrantz, *et al.* 78] Rosenkrantz, D.J., Stearns, R.E., and Lewis, P.M. : System Level Concurrency Control for Distributed Database Systems, *ACM Trans. on Database Systems*, Vol.3, No.2, pp.178–198, 1978.

[Smith 80] Smith, R.G. : The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solvers, *IEEE trans. on Computers*, Vol.29, No.12, pp.1104–1113, 1980.

[Waltz 75] Waltz, D. : Understanding Line Drawing of Scenes with Shadows, in P.H. Winston, editor, *The Psychology of Computer Vision*, pp.19-91, McGraw-Hill, 1975.

[Winston 77] Winston, P.H. : *Artificial Intelligence*, Addison-Wesley, 1975.

Constraint Satisfaction in a Connectionist Inference System

(Extended Abstract)

Steffen Hölldobler*
Fakultät für Informatik, Universität Dortmund
Postfach 500500
D-4600 Dortmund 50, Germany

Constraint satisfaction can be regarded as a technique for solving satisfiability problems in a restricted logical system. Though only constants, variables and Horn clauses are allowed, constraint satisfaction systems solve quite impressive problems (e.g. [van Hentenryck, 1989]). For many reasons – like the natural representation of constraint satisfaction problems as graphs and the solution of these problems by local and global propagation techniques – constraint satisfaction is often considered as a good candidate for the exploitation of parallelism. There are some limitations though. For example, Kasif [1990] has shown that the arc consistency problem is logspace-complete. On the other hand we should keep in mind that Kasif's result describes a worst case behavior. It still remains to be seen whether average practical problems reveal such bad manners.

But which technique shall we use if we want to parallelize constraint satisfaction? The results of Pinkas [1990] point into a possible direction. Pinkas has formally proved that the satisfiability problem of propositional logic is equivalent to the problem of finding a global minima of an energy function. In fact, since each constraint satisfaction problem can be presented as a propositional formula we can use Pinkas' formalism and construct an energy function whose global minima correspond precisely to the solutions of the constraint satisfaction problem and vice-versa. It is well-known that such energy functions can be implemented by artifical neural nets (see e.g. [Hinton *et al.*, 1986]). For example, we could design a Hopfield network and apply gradient descent to find a minima [Hopfield, 1982]. Alternatively, we could construct a Boltzman machine and try to find a minima by simulated annealing [Hinton *et al.*, 1986]. However, there is a problem. Both techniques, gradient descent as well as simulated annealing[1] do not guarantee that they find a global minima. They may as well get stuck in a local minima, which does not correspond to a solution of the constraint satisfaction problem. How often does this happen? There is no general answer and we have to look at special examples. One of these is the travelling salesman problem. In their initial paper concerning a connectionist solution of this problem, Hopfield and Tank [1985] reported that in only about 30% of

*on leave from FG Intellektik, FB Informatik, TH Darmstadt, Germany

[1]Simulated annealing may find a global minimum if the "temperature" is lowered in infinitesimal small steps, which is impracticable.

all cases the system did not succeed in finding a good solution. Such a good solution must not be a global minima, but close to a global minima such that the salesman has not to take a significant detour. However, reimplementations revealed that in up to 90% of all the cases the system failed to find acceptable solutions [Wilson and Pawley, 1988; Kamgar-Parsi and Kamgar-Parsi, 1990].

What are the alternatives? If we want to stick to the connectionist realm and to massive parallelism then we may employ the propagation of activation metaphor. Here, we are not searching for a global minima, rather activation is spread through a network of neurally-inspired units and the result of the computation is determined by the activation of certain output units. The operations performed as well as the messages sent by a unit are quite simple in a truly connectionist system. The messages contain almost no information. All knowledge is encoded in the connections between units. Such systems differ considerably from conventional parallel architectures. For an introduction into connectionist systems see eg. [Feldman and Ballard, 1982].

Shastri & Ajjanagadde [1990] have built a connectionist reasoning system for a limited class of first-order formula which exhibits impressive characteristics. The number of units used in the system is linear to the size of the formula or knowledge base and the time to answer or to respond to a query is bound by the depth of the search space. This optimal time behavior can be achieved by investigating all branches of the search space in parallel. Unfortunately, Shastri's & Ajjanagadde's system cannot be applied to solve constraint satisfaction problems due to their restrictions. In particular, the system does generally not guarantee that multiple occurrences of a variable are instantiated with the same term. In other words, the variable binding problem is not solved consistently. But if we consider Mackworth's [1977] formalization of constraint satisfaction, then this is precisely one of the problems we have to solve.

Cooper & Swain [1988] have designed a connectionist constraint satisfaction system. However, they limit their attention to arc consistency. In other words, they can handle only problems which can be solved by local constraint propagation. For each variable-value pair their system contains an initially active unit. Arc consistency is encoded as support for these units. A unit for the variable x and value a remains active as long as this value is supported by a respective constraint between x and each other variable in the system. Otherwise, the unit is deactivated. Guesgen [1990] has extended Cooper's & Swain's approach such that general constraint satisfaction problems can be solved. He uses a kind of Gödel numbering to encode path information in the potential of the variable-value units. To obtain this information the units must be able to pass Gödel numbers as messages and to compute greatest common divisors as well as least common multiples. This poses a certain problem since such complex messages and operations are considered to be biologically implausable. Guesgen refers to this problem by comparing his approch with Lange & Dyer's [1989] Robin, where signatures are used to encode constants and are passed as messages. However, the set of constants in Robin is usually quite small compared to the possible Gödel numbers and, hence, a distributed or localist representation of signatures complying with truly connectionist models is much easier to achieve (and has been done in the latest version of Robin) as with Gödel numbers.

CHCL is a connectionist inference system for Horn logic with limited resources [Hölldobler, 1990c; Hölldobler, 1990a]. The system is based on a connectionist unifi-

cation algorithm [Hölldobler, 1990b] and uses Bibel's connection method [Bibel, 1987]. There, a *connection* is a link between complementary literals. We know from [Bibel, 1987] or [Stickel, 1987] that a proof for a formula is found if we can identify a *spanning* and *complementary* set of connections. Informally, the spanning conditions ensures that the connections in the set form a complete proof of the formula on the propositional level obtained by omitting all arguments of predicates. Such a spanning set is complementary iff all connected literals are simultaneously unifiable. The search space can be compressed with the help of *reduction techniques*, which are typically applied linearly in time and space. The technique of interest as far as this note is concerned is the removal of *isolated* connections [Bibel, 1988a]. A connection is *isolated* iff the connected literals are not engaged in any other connection or the connected literals are ground or one of the connected literals is ground and the other one is not engaged in any other connection. Literals in isolated connections can be unified and under certain conditions the clauses containing these literals can be replaced by their resolvents. CHCL reduces a formula, generates the spanning sets and simultaneously unifies all connected literals in such a set.

From its expressive power CHCL can easily solve constraint satisfaction problems. But CHCL is not specifically designed for handling constraints and thus does not make use of the special features found in constraint satisfaction problems. It is the goal of this note to show

- how a simple modification of Mackworth's [1977] formalism allows to check in parallel, that the domain constraints are satisfied,
- how binary constraints which violate the domain constraints can be removed from the database in parallel,
- how a simple modification of CHCL and, thereby, of the underlying connection method allows to check for arc consistency in parallel, and
- how constraint satisfaction problems can be solved in a truly connectionist setting.

As an example consider a simple constraint network with variables x_1, x_2, and x_3, whose domains are $\{a, b\}$, $\{a, b, c\}$, and $\{b, c\}$, respectively. These variables are constraint by the relations $\langle a, a\rangle$, $\langle a, b\rangle$ between x_1 and x_2, $\langle b, b\rangle$, $\langle c, c\rangle$ between x_2 and x_3, and $\langle a, b\rangle$, $\langle a, c\rangle$ between x_1 and x_3. The domain constraints are expressed as simple facts. In our example we obtain the *domain facts*

$$D_1(a) \quad D_2(a) \quad D_3(b) \quad D_1(b) \quad D_2(b) \quad D_3(c) \quad D_2(c).$$

The constraints between variables are stated as binary predicates. However, these binary predicates are conditioned in that the values for the variables have to be elements of the domain of the respective variable. In our example we find the *constraint rules*

$$\begin{aligned}
C_{12}(a,a) &\Leftarrow D_1(a) \wedge D_2(a)\\
C_{12}(a,b) &\Leftarrow D_1(a) \wedge D_2(b)\\
C_{13}(a,b) &\Leftarrow D_1(a) \wedge D_3(b)\\
C_{13}(a,c) &\Leftarrow D_1(a) \wedge D_3(c)\\
C_{23}(b,b) &\Leftarrow D_2(b) \wedge D_3(b)\\
C_{23}(c,c) &\Leftarrow D_2(c) \wedge D_3(c).
\end{aligned}$$

We are interested in the values for the variables x_1, x_2, and x_3 such that all constraints are simultaneously fulfilled. In other words, we want to know whether there exists a substitution for the variables in

$$\Leftarrow C_{12}(x_1, x_2) \wedge C_{13}(x_1, x_3) \wedge C_{23}(x_2, x_3)$$

such that the respective instances of $C_{12}(x_1, x_2)$, $C_{13}(x_1, x_3)$, and $C_{23}(x_2, x_3)$ are logical consequences of the previously mentioned facts and rules. As the only difference to Mackworths' formalism we have moved the domain conditions from the goal clause to the constraint rules. This seems to be a minor difference. Obviously, the expressive power of the system is not affected. But as we will see in the sequel it is the basis for our connectionist constraint satisfaction system. As a side-effect it also allows to eliminate in a reduction step those constraint rules which specify constraints whose values are not in the domain of the respective variables. Note that all connections between the conditions of the constraint rules and the constraint facts are isolated. Replacing the constraint rules by their resolvents yields

$$\begin{array}{l} C_{12}(a, a) \\ C_{12}(a, b) \\ C_{13}(a, b) \\ C_{13}(a, c) \\ C_{23}(b, b) \\ C_{23}(c, c) \end{array}$$

which is precisely the formalization used in [Bibel, 1988b]. As described in [Bibel, 1988b] any constraint satisfaction problem can be transformed into this representation in linear time and space and we have done just this in the previous paragraphs. One should observe that, if we had a constraint rule like

$$C_{12}(c, c) \Leftarrow D_1(c) \wedge D_2(c)$$

then none of the connections between the domain facts and $D_1(c)$ is unifiable and, hence, the rule would be *useless*. This fact is detected in CHCL by using a kind of *weak unification* [Eder, 1985]. The unsolvability of all connections with $D_1(c)$ is propagated and, consequently, all connections with $C_{12}(c, c)$ will be unsolvable. This essentially removes the rule from the data base. One should observe that this check is done for all rules simultaneously. For more details see [Hölldobler, 1990a].

Let us point out that constraint satisfaction uses only constants and variables whereas CHCL as an inference system for Horn clause logic can deal with general first-order terms. Since no function symbols are involved, unification is no longer logspace-complete but parallelizable in any case [Dwork *et al.*, 1984]. Moreover, an inspection of the above example and a simple generalization shows that the unification problems solved within constraint satisfaction are always matching problems (ie. one of the involved terms does not contain variables). The connectionist unification algorithm built into CHCL can match terms in 2 steps and this is independent of the size of the matching problem [Hölldobler, 1990b]. Hence, the test whether a spanning set is complementary is performed in two steps. This speed-up is an inherent feature of CHCL.

We now come back to the constraint satisfaction procedure as described in [Bibel, 1988b]. Bibel applies a reduction technique called *database* (or DB-) reduction. He transforms the problem into a sequence of natural joins. We cannot repeat the details of this transformation in this note but we may briefly illustrate what happens in our example. By denoting the relation between the variables x_i and x_j by C_{ij} we find the solution to our example problem by computing

$$(C_{12} \bowtie C_{13}) \bowtie C_{23},$$

where $\bowtie$ denotes the natural equi-join (see e. g. [Maier, 1983]). Thus the problem of solving constraint satisfaction problems is transferred from the deductive part of a reasoning system to a database system. Bibel discusses in his paper various methods for computing sequences of equi-joins but he views this mainly as a database problem.

There is a certain gap in Bibel's approach. He does not deal with arc consistency. As the experienced reader may have noticed, the running example can be completely solved by local operations which ensure arc consistency. Formally, *arc consistency* states that a variable x may have value a if for each other variable y there is a value b such that $\langle a, b\rangle$ is a valid constraint between x and y, viz. $C_{xy}(a, b)$ is true. In other words, any variable y has to support each value for x. The arc consistency conditions allows to remove any value from the domain of x which is unsupported. This in turn may result in further unsupported values for the variables in the constraint network and, consequently, to the removal of these values. Though Bibel mentions Mackworth's [1987] observation that arc-consistency is a particular sequence of semi-joins, he does not address arc-consistency as a means to reduce the search space. Since he transforms (viz. reduces) a global constraint satisfaction problem into a sequence of natural equi-joins, it is the task of the database system to evaluate these equi-joins as fast and efficient as possible. However, I am not aware of a database system that uses semi-join operations as a reduction technique for sequences of equi-joins. Rather, semi-join operations are viewed as means to reduce the amount of communication in a distributed database system [Ullman, 1989].

Whereas Bibel solves global consistency problems using standard reduction techniques in his connection method without changing the calculus we will show how slight changes in the calculus achieve arc consistency. The changes are especially minor as far as the connectionist realization within CHCL is concerned. But let us first have a look at the connection method and arc consistency. The following two step algorithm AC realizes arc consistency.

AC Algorithm

1. For all i let S_{ij} be the set of values in the domain of x_i which are supported by x_j. Eliminate all atoms $D_i(c)$ if there is a j such that $c \notin S_{ij}$.

2. If step 1 results in useless constraint rules then eliminate these rules and goto step 1 else stop.

Applying the AC Algorithm to our running example we are left with the domain facts

$$D_1(a) \quad D_2(b) \quad D_3(b)$$

and the constraint rules

$$
\begin{array}{lcl}
C_{12}(a,b) & \Leftarrow & D_1(a) \wedge D_2(b) \\
C_{13}(a,b) & \Leftarrow & D_1(a) \wedge D_3(b) \\
C_{23}(b,b) & \Leftarrow & D_2(b) \wedge D_3(b).
\end{array}
$$

Recalling the initial query

$$\Leftarrow C_{12}(x_1,x_2) \wedge C_{13}(x_1,x_3) \wedge C_{23}(x_2,x_3)$$

we observe that all remaining connections in the formula are now isolated and can be evaluated simultaneously in one step. The formula collapses to the empty clause and we obtain the desired answer substitution

$$\{x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow b\}.$$

Proposition 1 *The AC algorithm transforms a given constraint satisfaction problem into an arc consistent constraint satisfaction problem such that both problems have exactly the same solutions.*

As previously mentioned, CHCL contains already a unit for each condition in a rule which will become active if the condition is found to be unsolvable. Furthermore, the unsolvability of conditions is propagated such that the respective rule becomes useless. Hence, all what remains to be done in order to ensure arc consistency within CHCL is to determine the sets S_{ij} and to eliminate all atoms $D_i(x)$ which are no longer supported. The sets S_{ij} are easily obtained from the constraint rules. Consider the constraints for the variables x_1 and x_2, viz.

$$
\begin{array}{lcl}
C_{12}(a,a) & \Leftarrow & D_1(a) \wedge D_2(a) \\
C_{12}(a,b) & \Leftarrow & D_1(a) \wedge D_2(b).
\end{array}
$$

S_{12} and S_{21} are simply the union of the arguments of the D_1 and D_2 predicates, respectively. Let us concentrate on the D_1 conditions. The D_2 conditions are treated analogously and simultaneously. There is a connection between each of the D_1 conditions and each domain fact for D_1. CHCL will detect which connections (viz. connected literals) are not unifiable. In the example these are all connection with the fact $D_1(b)$, which indicates that b is unsupported (viz. b is not in S_{12}). It is easy to extend CHCL to record this fact. All we need is an additional unit[2] for each domain fact. This unit will be called *unsupported* and is active iff the corresponding domain value is unsupported. Consequently, we have to remove the fact $D_1(b)$. A fact or a rule is essentially removed if all connections with the fact or the conclusion of the rule are unsolvable. CHCL contains already a unit for each connection which indicates whether the connection is unsolvable or not. Hence, all we have to do is to provide a link between the *unsupported* unit of $D_1(b)$ to the *unsolvable* unit of each connection with $D_1(b)$ such that the *unsolvable* units are activated as soon as the *unsupported* unit is active.

[2]More precisely, an OR-of-AND unit in the terminology of [Feldman and Ballard, 1982], where each AND-site corresponds to the constraint between the variable x_1 and another variable.

With the presented modifications CHCL becomes a connectionist inference system for constraint satisfaction which exploits the maximum parallelism inherent in local consistency operations in the sense that all parallalizable operations for checking the domain constraints as well as arc consistency are performed in parallel. Global consistency is computed by sequentially generating spanning sets of connections and, then, simultaneously unifying all connected literals in each spanning set. This can be illustrated if we add the constraint $\langle b, c\rangle$ to the set of constraints between x_2 and x_3. Formally, we have to extend our formula by

$$C_{23}(b, c) \Leftarrow D_2(b) \wedge D_3(c).$$

After the application of AC algorithm we are left with the domain facts

$$D_1(a) \quad D_2(b) \quad D_3(b) \quad D_3(c),$$

the constraint rules

$$\begin{array}{lcl} C_{12}(a,b) & \Leftarrow & D_1(a) \wedge D_2(b) \\ C_{13}(a,b) & \Leftarrow & D_1(a) \wedge D_3(b) \\ C_{13}(a,c) & \Leftarrow & D_1(a) \wedge D_3(c) \\ C_{23}(b,b) & \Leftarrow & D_2(b) \wedge D_3(b) \\ C_{23}(b,c) & \Leftarrow & D_2(b) \wedge D_3(c), \end{array}$$

and the query

$$\Leftarrow C_{12}(x_1, x_2) \wedge C_{13}(x_1, x_3) \wedge C_{23}(x_2, x_3).$$

The subgoal $C_{12}(x_1, x_2)$ is engaged in only one connection and, hence, is isolated. Furthermore, all connections involving domain facts are isolated. CHCL will evaluate these connection in parallel which results in the reduced query

$$\Leftarrow C_{13}(a, x_3) \wedge C_{23}(b, x_3)$$

and the reduced clauses

$$\begin{array}{l} C_{13}(a,b) \\ C_{13}(a,c) \\ C_{23}(b,b) \\ C_{23}(b,c). \end{array}$$

There are four connections left which can be combined to four spanning sets. CHCL generates and tests the spanning sets sequentially. However, the members of a spanning set are determined in parallel. In fact, for each constraint satisfaction problem, a spanning set is generated in 2 steps. The invocation of the unification algorithm requires 1 step and the unification (viz. matching) of all connected literals is performed in 2 steps. Hence, the example is solved after $5 \times 4 = 20$ steps with the two possible answers $\{x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow b\}$ and $\{x_1 \leftarrow a, x_2 \leftarrow b, x_3 \leftarrow c\}$. This is considerably faster than sequential implementations such as CONSAT [Güsgen, 1989], where the global consistency is computed via backtracking or tagging. There, to compute a solution the constraint graph has to be searched sequentially.

The process for determining global consistency is not particularly adaptated to constraint satisfaction problems. The system also does not make use of domain knowledge, which is known to reduce the search space considerably [Cooper and Swain, 1988]. These and other improvements are envisioned in future work. CHCL is currently being implemented at the International Computer Science Institute in Berkeley, California.

Acknowledgement: I would like to thank Hans-Werner Guesgen and Walter Hower for the intensive discussion which helped considerably to improve the ideas presented in this note.

References

[Bibel, 1987] W. Bibel. *Automated Theorem Proving.* Vieweg Verlag, Braunschweig, second edition, 1987.

[Bibel, 1988a] W. Bibel. Advanced topics in automated deduction. In Nossum, editor, *Fundamentals of Artificial Intelligence II.* Springer, 1988.

[Bibel, 1988b] W. Bibel. Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence*, 35:401–413, 1988.

[Cooper and Swain, 1988] P. R. Cooper and M. J. Swain. Parallelism and domain dependence in constraint satisfaction. Technical Report 255, Computer Science Department, Univ. of Rochester, 1988.

[Dwork *et al.*, 1984] C. Dwork, P. C. Kannelakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.

[Eder, 1985] E. Eder. Properties of substitutions and unifications. *Journal for Symbolic Computation*, 1:31–46, 1985.

[Feldman and Ballard, 1982] J. A. Feldman and D. H. Ballard. Connectionist models and their properties. *Cognitive Science*, 6(3):205–254, 1982.

[Güsgen, 1989] H. W. Güsgen. *CONSAT: A System for Constraint Satisfaction.* Pitman, 1989.

[Güsgen, 1990] H. W. Güsgen. A connectionist approach to symbolic constraint satisfaction, 1990.

[Hinton *et al.*, 1986] G. E. Hinton, J. L. McClelland, and D. E. Rumelhart. Distributed representations. In Rumelhart, McClelland, and the PDP Research Group, editors, *Parallel Distributed Processing*, chapter 3. MIT Press, 1986.

[Hölldobler, 1990a] S. Hölldobler. CHCL - A connectionist inference system for a limited class of Horn clauses based on the connection method. Technical Report TR-90-042, International Computer Science Institute, 1990.

[Hölldobler, 1990b] S. Hölldobler. A structured connectionist unification algorithm. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 587–593, 1990. A long version appeared as Technical Report TR-90-012, International Computer Science Institute, Berkeley, California.

[Hölldobler, 1990c] S. Hölldobler. Towards a connectionist inference system. In *Proceedings of the International Symposium on Computational Intelligence*, 1990.

[Hopfield and Tank, 1985] J. J. Hopfield and D. W. Tank. Neural computation of decisions in optimization problems. *Biological Cybernetics*, 52:141 - 152, 1985.

[Hopfield, 1982] J. J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. In *Proceedings of the National Academy of Sciences USA*, pages 2554 - 2558, 1982.

[Kamgar-Parsi and Kamgar-Parsi, 1990] B. Kamgar-Parsi and B. Kamgar-Parsi. On problem solving with hopfield neural nets. *Biological Cybernetics*, 62:415 - 423, 1990.

[Kasif, 1990] S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, 45(3):275–286, 1990.

[Lange and Dyer, 1989] T. E. Lange and M. G. Dyer. Frame selection in a connectionist model of high-level inferencing. In *Proceedings of the Annual Conference of the Cognitive Science Society*, pages 706–713, 1989.

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

[Mackworth, 1987] A. Mackworth. Constraint satisfaction. In Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 205–211. John Wiley & Sons, 1987.

[Maier, 1983] D. Maier. *The Theory of Relational Databases.* Pitman, 1983.

[Pinkas, 1990] G. Pinkas. The equivalence of energy minimization and propositional calculus satisfiability. Technical Report WUCS-90-03, Washington University, 1990.

[Shastri and Ajjanagadde, 1990] L. Shastri and V. Ajjanagadde. An optimally efficient limited inference system. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 563–570, 1990.

[Stickel, 1987] M. E. Stickel. An introduction to automated deduction. In W. Bibel and P. Jorrand, editors, *Fundamentals of Artificial Intelligence*, pages 75 - 132. Springer, 1987.

[Ullman, 1989] J. D. Ullman. *Principles of database and knowledge-base systems*, volume II of *The New Technologies.* Computer Science Press, 1989.

[van Hentenryck, 1989] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[Wilson and Pawley, 1988] G. V. Wilson and G. S. Pawley. On the stability of the travelling salesman problem algorithm of hopfield and tank. *Biological Cybernetics*, 58:63 - 70, 1988.

DIRECTED CONSTRAINT NETWORKS:

A RELATIONAL FRAMEWORK FOR CAUSAL MODELING

Rina Dechter
Information & Computer Science
University of California, Irvine
Irvine, CA. 92717
rdechter@ICS.UCI.EDU

Judea Pearl
Cognitive Systems Laboratory
Computer Science Department
UCLA, Los Angeles, CA. 90024
judea@cs.ucla.edu

Abstract

Normally, constraint networks are undirected, since constraints merely tell us which sets of values are compatible, and compatibility is a symmetrical relationship. In contrast, causal models use directed links, conveying cause-effect asymmetries. In this paper we give a relational semantics to this directionality, thus explaining why prediction is easy while diagnosis and planning are hard. We use this semantics to show that certain relations possess intrinsic directionalities, similar to those characterizing causal influences. We also use this semantics to decide when and how an unstructured set of symmetrical constraints can be configured so as to form a directed causal theory.

Area: Automated Reasoning / Constraints Satisfaction, or
Qualitative Reasoning / Causality

Declaration: This paper is not currently under review for a journal or another conference, nor will it be submitted during IJCAI's review period.

DIRECTED CONSTRAINT NETWORKS:

A RELATIONAL FRAMEWORK FOR CAUSAL MODELING †

1. Introduction

Finding a solution to an arbitrary set of constraints is known to be an NP-hard problem. Yet certain types of constraint systems, usually those describing causal mechanisms, manage to escape this limitation and permit us to construct a solution in an extremely efficient way. Consider, for example, the task of computing the output of a circuit consisting of a large number of logical gates. In theory, each gate is merely a constraint that forbids certain input-output combinations from occurring, and the task of computing the output of the overall circuit (for a given combination of the circuit inputs) is equivalent to that of finding a solution to a set of constraints. Yet contrary to the general constraint problem, this task is remarkably simple; one need only trace the flow of causation and propagate the values of the intermediate variables from the circuit inputs down to the circuit output(s). This forward computation encounters none of the difficulties of the general constraint-satisfaction problems, thus exemplifying the simplicity inherent to causal predictions.

The aim of this paper is to identify and characterize the features that render this class of problems computationally efficient, thus explaining some of the reasons that causal models are so popular in the organization of human knowledge. Note that this efficiency is asymmetric; it only characterizes the forward computation, but fails to hold in the backward direction. For instance, the problem of finding an input combination that yields a given output (a task we normally associate with planning or diagnosis) is as hard as any constraint satisfaction problem. Thus, the second aim of our analysis is to explain how a system of constraints, each defined in terms of the totally symmetric relationship of compatibility, can give rise to such profound asymmetries as those attributed to cause-effect or input-output relationships. At first glance, we might be tempted to

† This work was partially supported by the National Science Foundation, Grant #IRI-8821444 and by the Air Force Office of Scientific Research, Grant #AFOSR-90-0136.

attribute the asymmetry to the functional nature of the constraints involved. However, functional dependency in itself cannot explain the directional asymmetry found in the analysis of causal mechanisms such as the logic circuit above. Imagine a circuit containing some faulty components, the output of which may attain one of *several* values. The constraints are no longer functional, yet the asymmetry persists; finding an output compatible with a given input is easy while finding an input compatible with a given output is hard. This asymmetry between prediction and planning seems to be a universal feature of all systems involving causal mechanisms [Shoham, 1988], a feature we must emulate in defining causal theories.

Our starting point is to formulate a necessary and sufficient condition for a system of constraints to exhibit a directional asymmetry similar to that characterizing causal organizations. Basically, the criterion is that there should exist an ordering of the variables in the system such that imposing constraints on later variables would not further constrain earlier variables. Intuitively, it captures the understanding that predictions are useless for diagnosis. Starting with this criterion as a definition of causal theories (Section 2), we show that it is tantamount to enabling **backtrack-free** search (for a feasible solution) along the natural ordering. We then explore methods of constructing causal specifications for a given relation, that is, specifications that permit objects from the relation to be retrieved backtrack-free along some ordering. Such methods are investigated along two dimensions: inductive and pragmatic. Along the inductive dimension (Section 3), we are given the tuples of some relation ρ, and we seek to represent this set of tuples by a causal theory that is as *simple* as possible. We provide a formal definition of simplicity and show that together with the insistence on backtrack-free predictions, it leads to a natural definition of *intrinsic directionality*, matching our perception of causal directionality in logical circuits and other physical devices.

Along the pragmatic dimension (Section 4), we start with an unordered collection of constraint specifications, which might represent some stable physical laws, and we seek an ordering of the variables such that the overall system constitutes a causal theory. Clearly, not every system of constraints can turn causal by a clever ordering of the variables. The criterion for the existence of such an ordering depends on both the nature of the constraints and the topology of the subsets of variables upon which the constraints are specified. Some constraint systems are amiable to causal ordering by virtue of their topology alone, *regardless* of the content of the individual constraints. These are called acyclic constraint systems, originally studied in the literature of relational databases, [Beeri et al 1983]. In contrast, Section 4 ascribes causal ordering to a more general set of topologies, but imposes special requirements on the character of the individual constraints.

Our basic requirement for a k-variable constraint to qualify as a description of a primitive causal mechanism, is that at least one set of $k-1$ variables must behave as **inputs** (or **causes**) relative to the remaining k^{th} variable (to be regarded as an **output** or an **effect**), in the sense that each value combination of these $k-1$ variables must be compatible with at least one value of the k^{th} variable. Additionally, in order for the system as a whole to act as a causal system, the constraints must be ordered in a way that prevents conflicts among the various outputs, hence, no two constraints should designate the same variable as an output. We provide effective procedures for: (1) deciding if such an ordering exists and, (2) identifying such ordering whenever possible. The ordering found can be used to facilitate search and retrieval, and are similar to those used to describe the operation of physical devices [Kuipers, 1984] [Iwasaki, 1986] [de-Kleer,1986] [Forbus, 1986]

2. Definitions and Preliminaries: Constraint Specifications and Causal Theories

Definition 1 (Constraint Specification): A **constraint specification** (CS) consists of a set of n variables $X = \{X_1, \ldots, X_n\}$, each associated with a finite

domain, $dom_1, \ldots, dom_n$, and a set of constraints $\{C_1, C_2, \ldots, C_t\}$ on subsets of X. Each **constraint** C_i is a relation on a subset of variables $S_i = \{X_{i_1}, \ldots, X_{i_j}\}$, namely, it defines a subset of the Cartesian product of $dom_{i_1} \times, \ldots, \times dom_{i_j}$. The **scheme** of a CS is the set of subsets on which constraints are defined, $scheme(CS) = \{S_1, S_2, \ldots, S_t\}$, $S_i \subseteq X$. A **solution** of a given CS is an assignment of values to the variables in X such that all the constraints in CS are satisfied. A constraint specification CS is said to define an **underlying relation** $rel(CS)$, consisting of all the solutions of CS.

Definition 2 (Causal Theories): Given a constraint specification CS, its underlying relation $\rho = rel(CS)$, and an ordering $d = (X_1, X_2, \ldots, X_n)$, we say that a CS is a **causal theory** (of ρ) relative to d if for all $i \geq 1$ we have

$$\Pi_{X_1, \ldots, X_i}(\rho) = \bowtie_{j: S_j \subseteq \{X_1, \ldots, X_i\}} C_j \qquad (1)$$

where $\Pi_{X_1, \ldots, X_i}(\rho)$ denotes the projection of ρ on $\{X_1, \ldots, X_i\}$, that is,

$$\Pi_{X_1, \ldots, X_i}(\rho) = \{x = (x_1, \ldots, x_i) \mid \exists \bar{x} \in \rho, \bar{x} \text{ is an extension of } x\}, \qquad (2)$$

and $\bowtie$ is the *join* operator. Any pair $<d, CS>$ satisfying (1) will be called a **causal theory** (of ρ).

Although condition (1) may seem hard to verify in practice, it nevertheless provides an operational definition for causal theories. To test whether a given CS is causal relative to ordering d, we need to find the set of solutions to the given CS, project back these solutions on the strings of variables $X_1, X_2, \ldots, X_i$, $1 \leq i \leq n$, then check whether each such projection coincides exactly with the set of solutions to a smaller CS, one consisting of only those constraints that are defined on variables taken from $\{X_1, \ldots, X_i\}$. In Section 4 we will show that certain types of specifications possess syntactic features that render them inherently causal, in no need of the elaborate test prescribed by (1). For example, the specifications of logic gates are always causal relative to orderings compatible with their interconnections in logic circuits. Similarly, linear inequalities and

propositional clauses, under certain conditions, can be assembled into causal theories by finding appropriate orderings of the variables.

From a conceptual viewpoint, Definition 2 can be given the following interpretation. If we view the variables $X_1, \ldots, X_i$ as past events, the variables $X_{i+1}, \ldots, X_n$ as future events, and the constraints as physical laws, then Eq. (1) asserts that the permissible set of past scenarios is not affected by laws that pertain only to future events. In other words, past events cannot be ruled out by considering their impact on future events. This interpretation is indeed at the very heart of the notion of causation, and is closely related to the principle of *chronological ignorance* described in [Shoham 1988], although Shoham's definition of causal theories insists on functional dependencies.

We shall now show that causal theories as defined by (1) yield a computationally effective scheme of encoding relations; it guarantees that the tuples in these relations can be generated systematically, without search, by simply instantiating variables along the natural ordering of the theory.

Definition 3 (Backtrack-free): We say that a CS is **backtrack-free** along ordering $d = (X_1, \ldots, X_n)$ if for every i and for every assignment $x_1, \ldots, x_i$ consistent with $\{C_j : S_j \subseteq \{X_1, \ldots, X_i\}\}$ there is a value x_{i+1} of X_{i+1} such that $x_1, \ldots, x_i, x_{i+1}$ satisfies all the constraints in $\{C_j : S_j \subseteq \{X_1, \ldots, X_{i+1}\}\}$. In other words, a CS is backtrack-free w.r.t. d if $rel(CS)$ can be recovered with no dead-ends along the order d.

Theorem 1: A constraint specification CS is backtrack-free along an ordering d if and only if it is causal relative to d.

Definition 4 (Dags and Families): Given a directed acyclic graph (dag) D, we say that an ordering $d = (X_1, \ldots, X_n)$ of the nodes in the graph **respects** D if all edges in D are directed from lower to higher nodes of d. A dag D defines a set of n **families** $F_1, \ldots, F_n$, each family F_i is a subset consisting of a **son** node, X_i, and all its **parent** nodes, P_i, which are those directed towards X_i in D.

Definition 5 (Characteristic dag): The **characteristic dag, D,** of the pair (d, CS) is constructed as follows: For each set S_j in *scheme(CS)*, designate the largest variable in S_j as a sink and direct the other variables in S_j towards it.

Figure 1 shows the characteristic dag of a CS defined on the subsets AD, DC, DEF, AB, BC, CF, along the ordering $d = (B, A, C, D, F, E)$.

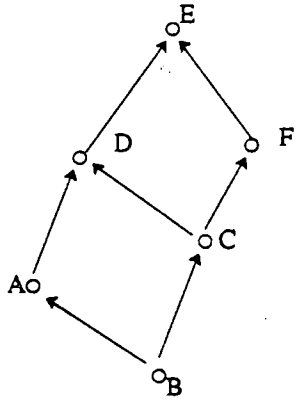


Figure 1: The characteristic dag of a CS

Lemma 1: If D is the characteristic dag of the pair (d, CS) then it is also the characteristic dag of (d', CS) whenever d' respects D and, furthermore, if $<d, CS>$ is a causal theory, then so is also $<d', CS>$. □

We can, therefore, characterize a given causal theory $<d, CS>$ by the pair $<D, CS>$ where D is the characteristic dag of (d, CS). Indeed, the prevailing practice in causal modeling is to use dags, not total orders, to describe the structure of causal organizations. Note that for $<D, CS>$ to be a causal theory, the families of D must form a partition of *scheme*(CS), because to comply with the construction of Definition 5, any set of variables that supports a single constraint must reside within at least one family of D.

Definition 6 (Causal model): Given a relation ρ and an arbitrary dag D, D is a **causal model** of ρ if there exists a constraint specification CS such that $<D, CS>$ is causal theory of ρ. In other words, if there exists a constraint specification CS defined on subfamilies of D such that $rel(CS) = \rho$ and such that CS is causal

relative to some ordering respecting D.

A causal model is a graph which merely designates qualitatively the existence of direct causal influences among sets of variables, but does not specify the nature of the influences.

3. Synthesizing Causal Theories and Uncovering Causal Directionality

Our ultimate goal is to construct causal theories for the information we possess. Since our formulation of causal theory consists of three elements: a relation ρ, a constraint specification CS, and a dag D, the natural question that arises is: Given any two of the three elements, can we find the third so as to form a causal theory (i.e., satisfying condition (1) while maintaining $\rho = rel(CS)$)? These questions will be explored in Sections 3 and 4.

Task 1: (decomposition) Given a relation ρ and an ordering d, find a causal theory for ρ along d.

Barring additional requirements, a causal theory can be obtained by a trivial construction. For instance, the complete dag generated by directing an edge from each lower variable to every higher variable is clearly a causal model of ρ, and the desired causal theory can be obtained by projecting ρ onto the complete families $F_i = \{X_1, X_2, \ldots, X_i\}$. We next present a scheme for constructing a causal theory on top of an **edge-minimal** model of ρ, that is, a dag D from which no edge can be deleted without destroying its capability to support a causal theory of ρ.

The algorithm that follows constructs an edge-minimal causal model of ρ.

build-causal-1 (ρ,d):
1. Begin
   2. For $i = n$ to 2 by -1 do:
   3. Find a minimal subset $P_i \subseteq \{X_1, \ldots, X_{i-1}\}$ such that
      $\Pi_{X_1, \ldots, X_{i-1}}(\rho) \bowtie \Pi_{P_i \cup \{X_i\}}(\rho) = \Pi_{X_1, \ldots, X_i}(\rho)$
   4. Return a dag D generated by directing an arc
      from each node in P_i towards X_i.
5. End.

To form a causal theory, we simply pair this dag with the projections of ρ on its families.

The construction above shows that a causal theory can be found for any arbitrary ordering. However, we will next show that certain orderings possess features that render them more natural for a given relation. It is these features, we conjecture, which give rise to the perception that certain relations possess "intrinsic" directionalities.

Definition 7 (Model Preference): A causal model D_2 is said to be at least **as expressive as** D_1, denoted $D_1 \leq D_2$, if for any causal theory $<D_1, CS_1>$ there exists a causal theory $<D_2, CS_2>$ such that $rel(CS_1) = rel(CS_2)$. A dag D is said to be a **minimal** causal model of ρ if it is not strictly more expressive than any other causal model of ρ.

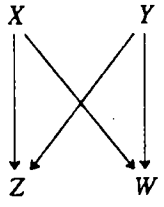
Definition 8 (Intrinsic Directionality): Given a relation ρ, a variable X is said to be a **direct cause** of variable Y, if there exists a directed edge from X to Y in all minimal causal models of ρ.

Example 1. Consider a relation ρ specified by the table of Figure 2(a). The table is small enough to verify that the dag in 2(b) is the only minimal causal model of ρ. For example, the arrow from X to Z cannot be reversed, because ρ cannot be expressed as a set of constraints on the families of the resulting dag, $\{YZ, ZX, XYW\}$. Adding an arc $Y \rightarrow X$ to the resulting dag would permit a representation of ρ (using the scheme $\{YZ, YZX, YXW\}$), but would no longer be minimal, being strictly more expressive than the one in 2(b). The causal theory

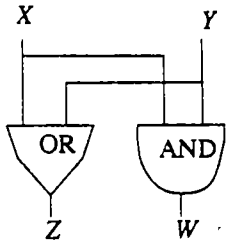
corresponding to the dag of 2(b) is shown in 2(c), matching our intuition about the causal relationships embedded in 2(a). Note that the same minimal model ensues (though not the same theory) were we to destroy the functional dependencies by adding the tuple 1100 to the table in 2(a).

X	Y	Z	W
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

(a)



(b)



(c)

Figure 2

Verma and Pearl [Verma, 1990] have used minimal model semantics to construct a probabilistic definition of causal directionality. They have also developed a proof theory which, under certain conditions provides efficient algorithms for determining causal directionality without inspecting the vast space of minimal models. Whether similar conditions exist in the relational framework remains an open problem.

Task 2: Given a constraint specification CS, find a dag D and a constraint specification CS', s.t. $<D, CS'>$ is a causal theory of $rel(CS)$.

This task can be solved by executing algorithm *Adaptive-consistency* [Dechter, 1987] along an arbitrary ordering d:

build-causal-2 (CS, d)
1. Begin
2. Execute *adaptive-consistency* w.r.t order d.
3. Take the graph induced by *adaptive consistency* and direct edges from lower to higher variables, Call this dag D.
4. For each variable, X_i, return the constraint C_i that *adaptive consistency* induces on the i^{th} family of D.
5. End

The resulting pair $<D, \{C_i\}>$ is a causal theory of CS. Algorithm *adaptive-consistency* is known to be exponential in the induced width W^* of *scheme*(CS) [Dechter, 1987], hence, it is a practical procedure only for sparse constraint topologies.

4. Finding Causal Ordering

In this section we characterize sufficient conditions under which causal theories can be assembled from pre-existing constraint specifications. Part of these conditions relate to the nature of the individual constraints; each must permit a causal relationship to exist between variables designated as *inputs* and that designated as the *output*. Additionally, to avoid conflicts in assembling the overall theory, we shall also insist that no two constraints designate the same variable as their output. Whether a given CS can comply with the latter restriction depends only on the topological property of *scheme*(CS) and is captured in the notion of an **ordered *CS***.

Definition 9: An **ordered constraint specification** (**OCS**) is a pair (D, CS) consisting of a dag D, and a special CS s.t. *scheme*(CS) is isomorphic to the families of D, namely, $C \in$ *scheme*(CS) iff $C = F_i$ for some i in D. The constraints (of which there are at most n) will be denoted by $C_{i_1}, \ldots, C_{i_t}$, $t \leq n$, each indexed by the son of the corresponding family.

For example, a CS with constraints $\{ADC, DEF, AB, BC, CF\}$ together with the dag of Figure 1 is clearly an OCS. However, if instead of ADC, we had two separate constraints, on AD and CD, this dag could no longer be paired with the CS to form an OCS and, in fact, no such dag exists.

Task 3: Given a CS find, whenever possible, a dag D s.t. (D, CS) is an OCS.

Procedure build-dag-3 (CS)
1. While *scheme* (CS) is non-empty do
2. If there is a subset $S \in$ *scheme* (CS) having a **lonely** variable X (i.e., one that participates in only one constraint), then
3. direct edges from all variables in S towards X, and remove S from *scheme* (CS).
4. else, return failure.
5. end while
6. return the dag D generated.

Theorem 2: Procedure **build-dag-3** (CS) returns a dag D if and only if (D, CS) is an OCS. Moreover, the dag returned is unique. □

Task 4: Given a CS, find, whenever possible, a dag D s.t. $<D, CS>$ is a causal theory of *rel* (CS).

In general, this task seems to require insurmountable amount of computations. The task becomes easier when the CS can be assembled in an OCS by the procedure above. Still, not every OCS pair (D, CS) corresponds to a causal theory $<D, CS>$ according to criterion (1). We next show that if the constraints residing in a given OCS meet certain conditions, then the OCS always yields a causal theory. Such constraints will be called *causal.*

Definition 10 (Causal constraints): A constraint, C, on a set of variables $U = \{X_1, \ldots, X_{k-1}, X_k\}$ is said to be **causal** with respect to a subset O of its variables if the following three conditions are met: *i).* Any assignment of values to $U-O$ is legal. Formally, if $O = \{X_{i_1}, \ldots, X_{i_{|O|}}\} \subset U$, then $\Pi_{U-O}(C) = dom_{i_1} \times, \ldots, \times dom_{i_{|O|}}$. *ii).* Let O_j denote the set $\{X_{i_j}\} \cup U-O$, then $C = \Pi_{O_1}(C) \bowtie, \ldots, \bowtie \Pi_{O_{|O|}}(C)$. In other words, C can be **losslessly**(1) decomposed into $|O|$ smaller constraints, each defined on $U-O$ plus a single variable from O. *iii).* O has no superset satisfying (*i*) and (*ii*). We say that $U-O$ and O are **inputs** and **outputs**, respectively, of the causal constraint C.

(1) A relation ρ is said to be *losslessly decomposed* into $\rho_1, \ldots, \rho_t$ if $\rho = \rho_1 \bowtie \rho_2 \bowtie, \ldots, \bowtie \rho_t$.

Example 2: Consider the constraint C specified in Example 1. The sets $\{X, Y\}$, $\{X\}$, $\{Y\}$, $\{Z\}$, and $\{W\}$ qualify as input sets ($U-O$) according to condition (i), since C permits all possible assignments to their constituents. However, only $\{X, Y\}$ qualifies as an input set by requirement (ii), since C can be losslessly decomposed only into the scheme $\{XYZ, XYW\}$. Hence, C is *causal* w.r.t. $O = \{Z, W\}$, and indeed, it matches the functional description of C as shown in Figure 2(c).

Definition 11 (Symmetric constraints): A causal constraint is said to be **symmetric** if it is causal with respect to each of its singleton variables.

For example, the constraint on $\{X, Y, Z\}$ given by the linear inequality $X + Y + Z \leq a$, is causal and symmetric, since any one of the three variables qualifies as an output, with the other two as inputs. It is easy to verify that linear equalities, e.g. $X + Y + Z = a$ and propositional clauses, e.g. $X \vee Y \vee Z$ are also symmetric.

Theorem 3: If in a given OCS, (D, CS), the constraint associated with each family F_i is causal w.r.t. X_i, then $<D, CS>$ is a causal theory. □

Corollary: An unordered set of *causal* constraints can be assembled into a causal theory if there is a dag D that will render it an OCS and if the sons in the families of D coincide with the output variables in the causal constraints. □

To test for these conditions, it is sufficient to run the **build-dag-3** algorithm and check if the returned dag satisfies the last condition of the corollary. Since **build-dag-3** runs in quadratic time, the entire construction can be accomplished in quadratic time.

A subclass of OCS's that is causal w.r.t. **any** specification of the constraints is the well known **acyclic** CS [Dechter, 1989] which is closely related to acyclic databases [Beeri, 1983]. It can be shown that models enforcing local consistency between adjacent constraints is sufficient for rendering any acyclic OCS backtrack-free, hence, a causal theory.

In many AI systems knowledge is expressed as a conjunction of propositional clauses, where each clause is a disjunction of literals. One can view each literal as a variable accepting one of two values, and each clause as a constraints on its literals.

Lemma 2: Every clause is a symmetric causal constraint w.r.t each of its singleton variables. □

Corollary: A set of clauses that forms an *OCS* is always satisfiable and, moreover, a satisfying assignment can be found in linear time. □

In case algorithm **build-dag-3** fails, we know that the specifications do not lend themselves to causal modeling by straightforward variable ordering. It is still feasible though that causal theories could be formed by treating clusters of variables as single objects. Such clusters were permitted, for example, in the causal ordering of Simon [Iwasaki, 1980], which was restricted to the case of linear equations. The basic **build-dag-3** algorithm can be used to identify promising candidates of variable clusters, and to assemble a more powerful type of causal theories than those treated in this paper.

5. Conclusions

This paper presents a relational semantics for the directionality associated with cause-effect relationships, explaining why prediction is easy while diagnosis and planning are hard. We used this semantics to show that certain relations possess intrinsic directionalities, similar to those characterizing causal influences. We also provided an effective procedure for deciding when and how an unstructured set of symmetrical constraints can be configured so as to form a directed causal theory.

These results have several applications. First, it is often more natural for a person to express causal relationships as directional, rather than symmetrical constraints. The semantics presented in this paper permits us to interpret and process

directional relationships in a consistent way and to utilize the computational advantages latent in causal theories. Second, the notion of intrinsic directionality suggests automated procedures for discovering causal structures in raw observations or, at the very least, for organizing such observations into structures that enjoy the characteristics of causal theories. Finally, the set of constraint specifications that can be configured to form causal theories constitutes another "island of tractability" in constraint satisfaction problems. The procedure provided for identifying such specifications can be used to order computational sequences in qualitative physics and scheduling applications.

References

[Beeri, 1983] Beeri, C., R. Fagin, D. Maier and M. Yannakakis, "On the Desirability of Acyclic Database Schemes," *Journal of ACM,* Vol. 30, No. 2, July, 1983, pp. 479-513

[de-Kleer, 1986] de-Kleer, J. and J.S. Brown, "Theories of causal ordering," *Artificial Intelligence,* Vol. 29, No. 1, 1986, pp. 33-62.

[Dechter, 1987] Dechter, R. and J. Pearl, "Network-based Heuristics for Constraint-Satisfaction Problems," *Artificial Intelligence,* Vol. 34, No. 1, December, 1987, pp. 1-38.

[Dechter, 1989] Dechter, R. and J. Pearl, "Tree Clustering for Constraint Networks," in *Artificial Intelligence,* 1989, pp. 353-366.

[Forbus, 1986] Forbus, K. and D. Gentner, "Causal reasoning about quantities," in *Proceedings of the Eighth Annual Meeting of the Cognitive Science Society,* Amherst, MA, 1986, pp. 196-207.

[Iwasaki, 1986] Iwasaki, Y. and H.A. Simon, "Causality in device behavior," *Artificial Intelligence,* Vol. 29, No. 1, 1986, pp. 3-32.

[Kuipers, 1984] Kuipers, B. "Common sense Reasoning about Causality: Deriving Behavior from Structure," *Artificial Intelligence,* Vol. 24 (1-3), 1984, pp. 169-203.

[Shoham, 1988] Shoham, Y., *Reasoning About Change,* Cambridge, Massachusetts: The MIT Press, 1988.

[Verma, 1990] Verma, T. and J. Pearl, "Equivalence and Synthesis of Causal Models," in *Proceedings, Sixth Conference on Uncertainty in AI,* Cambridge, Massachusetts, July 27-29, 1990, pp. 220-227.

The Role of Compilation in Constraint-Based Reasoning

Yousri El Fattah (fattah@ics.uci.edu)
Paul O'Rorke (fattah@ics.uci.edu)

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

Abstract

Given a set of constraints and a set of assumptions, we can propagate a premise set through the constraints to determine all possible inferences. Those inferences allow us to answer queries regarding the consistency of the premise set, the assumptions underlying every inference, or the sets of assumptions (nogoods) leading to contradictions. We describe a constraint propagation system initially developed for model-based diagnosis [9]. The system maintains labels of all assumptions underlying every inference, in a manner similar to an ATMS. Unlike ATMS, the justifications and premises in our TMS are represented as first-order predicate formulae, which include variables. Our TMS accumulates macro rules so as to speed up answers to queries for different instantiations of the premise set. In one version, the compilation is done in advance, while in the other, is done when answering queries for a specific premise set, in a manner similar to explanation-based learning (EBL) [8, 20]

1 Introduction

Constraint propagation is of central importance in search-based problem solving. Value inference is one kind of constraint propagation, where values for unassigned variables are deduced from the values already assigned. This technique has been used in various diagnosis algorithms [5, 2]. It is also the basis of the CONSTRAINTS language [19].

Truth Maintenance Systems (TMS) can be viewed as a kind of constraint propagation. A TMS-based problem solver consists of two components: an inference engine and a TMS. The TMS's task is to determine what is and what is not believed, and the inference engine's task is to make inferences about the domain. In ATMS [3] all such inferences are recorded and communicated to the ATMS as *justifications*. Every problem solving hypothesis is

communicated to the ATMS as an *assumption*. A set of assumptions is an *environment* and the set of all data derivable from the assumptions is the *context* of the environment. The ATMS associates with every datum a set of minimal environments from which it is derivable. The set is the *label* of a datum. The task of the ATMS is to guarantee that each label of each node is consistent, sound, complete, and minimal with respect to the justifications. A task involving n assumptions has 2^n environments, and at most 2^n contexts. The ATMS is required to tell when a context becomes inconsistent and whether a node holds in a particular context.

In an ATMS, the label generation algorithm actually calculates the set of prime implicates for the set of justifications [17]. Provan [15] has shown that ATMS label manipulation is of exponential complexity in the worst case. This is because there can be an exponential number of minimal support clauses (prime implicates) for a set of input clauses.

McAllester's [13] justification-based TMS (JTMS) is a single-context system. In a single-context system the TMS maintains for the problem solver only one consistent subset (a context) of the data that has been passed to the TMS, whereas the multiple context system (like ATMS) provide a facility for determining contexts dynamically, without enforcing the usage of any particular one. Although McAllester's JTMS is efficient and tractable, the multiple-context feature of ATMS is useful in device diagnosis if one wants to find the minimal number of possible faults that explains a given observed behavior [5]. In single-fault diagnosis, however, the full-generality of the ATMS is not needed.

Reiter and de Kleer [17] compares an interpreted versus compiled approach to a clause management system (CMS) — a generalization of de Kleer's ATMS. In the compiled approach, the CMS does not store the clauses transmitted to it by the "Reasoner" (as in the interpreted approach) but rather the prime implicants of these clauses. The reward for this is that the cost of queries becomes cheap.

Freuder [10] described a method for constraint propagation by synthesizing new ones. The method constitutes a sort of compilation process using a dynamic programming technique. That compilation is a form of learning in advance by means of constraints preprocessing. Inducing all possible constraints may involve a procedure which is exponential both in time and space [10]. Dechter [6] proposed a method of learning while searching that consisted of identifying minimal conflict sets contributing to dead-ends in a a backtrack search. Constraint compilation also appears in the interpretation of constraint logic programming, as described by the CLP interpreter in Cohen [1]. Van Hentenryck's [11] version of Prolog called Chip (for constraint handling in Prolog) combines automatic backtracking with automatic constraint propagation. Dechter [7] defines and compares the performance of various schemes suggested in the areas of constraint satisfaction problems, logic programming, and truth maintenance systems for enhancing the performance of the backtracking algorithms. Those schemes are classified into two types: those that are employed *in advance* of performing the search, and those that used *dynamically* during search.

All truth maintenance systems manipulate proposition symbols and relationships be-

tween proposition symbols. They address the search problem for a specific premise set. Should the problem premises change, the inference and search process are repeated, though the problem may have the same or "similar" solution as the ones that were solved before. In a recent work [9], we addressed this problem in the context of multiple fault diagnosis. Instead of propagating the assumptions for each propositional datum, we propagate the assumptions for a generalization of the datum. The generalization is such that the set of assumptions for each datum remain valid for any instantiation of the premise set. Our approach is similar to explanation-based learning [8, 20], although it shares common features with ATMS and constraint logic programming. We present in this paper a procedural description of our approach. We hope that this paper will facilitate further integration of ideas from explanation-based learning, constraint logic programming, and truth maintenance systems.

2 Motivation

An Assumption-Based Truth Maintenance system (ATMS)[3] receives from a deductive module a set of pairs (assertion, justification). The pairs correspond to propositional Horn clauses. The elements of a base of facts maintained by the ATMS are denoted by: (assertion, list of justifications, label). The label of an assertion A is a set of environments denoted by $\{E_1, E_2, \ldots, E_p\}$. An environment E is a set of basic assumptions denoted by $\{H_1, H_2, \ldots, H_k\}$. A *basic assumption* H is an assertion specified as such by the user. The deductive module may transmit pairs (contradiction, justification) to ATMS. The environments of a contradictory assertion are called *nogood sets*. Every superset of a contradictory environment is itself contradictory. The task of the ATMS is to calculate and update the *label* of each assertion. Labels should be sound, complete, consistent, and minimal [4].

The ATMS may only be used to answer questions concerning instantiated cases. For example, the nogood sets will be applicable only to the propositional premises communicated by the deductive module. For another instantiation of premises, ATMS will have to calculate and update the labels of various assertions anew.

We borrow the following example from [18]. The example involves the following base of rules:

$$student(X) \wedge H_1(X) \rightarrow young(X) \quad (1)$$

$$young(X) \wedge H_2(X) \rightarrow single(X) \quad (2)$$

$$student(X) \wedge parent(X) \wedge H_3(X) \rightarrow married(X) \quad (3)$$

Rule 1 says: "students are young"; Rule 2: "young people are single"; Rule 3: "students who have children are married". $H_i(X)$ is the assumption that rule i applies to individual

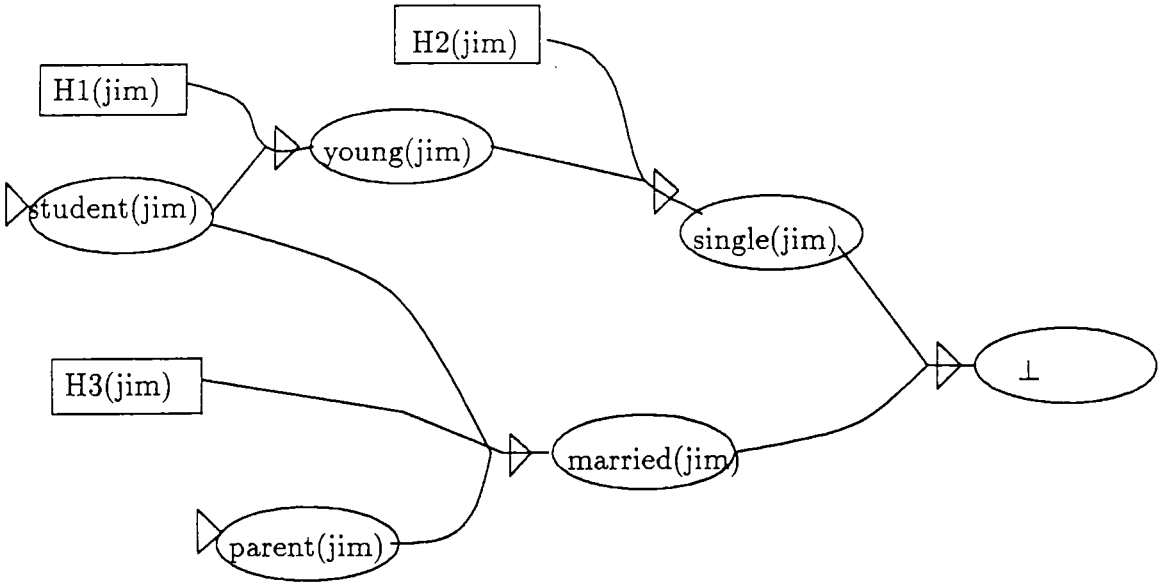


Figure 1: A contradiction

X $(i = 1, 2, 3)$. Consider the set of premises: $student(jim)$, $parent(jim)$ The deductive engine transmits the assertions:

$$young(jim), single(jim), married(jim)$$

along with their justification. The deductive engine also transmits the contradiction with the justification $married(jim) \wedge single(jim)$. The state of the knowledge base is depicted in figure 1. Based on this, the ATMS concludes that $\{H_1(jim), H_2(jim), H_3(jim)\}$ is a nogood for the given set of premises.

Explanation-based learning (EBL) [8, 20] aims at generalizing explanations (justification) of assertions derived by a deductive module in order to speed-up the derivation of "similar" assertions. The propositional nature of ATMS means that for a new set of premises such as

$$student(bob), parent(bob)$$

the work is repeated, and ATMS concludes that $\{H_1(bob), H_2(bob), H_3(bob)\}$ is a nogood set.

For the example above our TMS following the jim example would conclude the following macro rules

$$student(X) \wedge parent(X) \rightarrow nogood([H_1(X), H_2(X), H_3(X)]) \quad (4)$$

3 Problem Statement

We pose our problem in terms of a constraint satisfaction problem (CSP) [12]. Assume the existence of a finite set I of variables $X = \{X_1, X_2, \ldots, X_r\}$, which take respectively their values from their finite domains $D_1, D_2, \ldots, D_r$ and a set of constraints. A constraint $c(X_{i_1}, X_{i_2}, \ldots, X_{i_k})$ between k variables from I is a subset of the Cartesian product $D_1 \times D_2 \times \ldots \times D_r$, which specifies which values of the variables are compatible with each other.

We are also given a set of assumptions $H = \{H_1, \ldots, H_r\}$, where assumption H_i is in conjuction with constraint C_i.

Constraints are represented as inference rules in the form:

$$Head \leftarrow Body\ \{Constraints\} \tag{5}$$

The rules specify what information on variable assignments can be deduced once some variable assignments are available. Assignments inferred correspond to what we call *conclusion* variables, while known assignments leading to the application of the constraints correspond to what we call *triggering* variables.

Example 1. *An adder constraint whose input variable assignments are SIn1, SIn2 and output variable assignment is SOut can be expressed as*

$$\begin{aligned} \neg AB(Adder) \quad \Rightarrow \quad & out(Adder) = SOut, in1(Adder) = SIn1, in2(Adder) = SIn2, \\ & \{SOut \equiv SIn1 + SIn2\} \end{aligned} \tag{6}$$

Adopting Reiter's convention [16], AB(c) is the assumption that component c is defective (behaves abnormally).

The problem is as follows. For a given initial set of premises: $\sigma = \{X_{i_1} = x_{i_1}, X_{i_2} = x_{i_2}, \ldots, X_{i_r} = x_{i_s}\}$, $s < r$, forming a partial assignment over X, we want to be able to answer queries such as:

1. Is the assignment σ consistent with the set of constraints?
2. If σ is inconsistent, what are the minimal *nogoods*?
3. For every deduced variable assignment, what are all the (minimal) assumptions under which it holds?

4 Value Inference

In order to answer the queries stated in section 3 we need to propagate the assignments of the premise set through the constraints, using all possible inference rules. In the process of

making value inferences, it is important to keep track of the underlying assumptions and avoid redundant or circular inferences. The process halts when all possible inferences are made. This is illustrated by the procedure PROPAGATE.

ALGORITHM **PROPAGATE**($PremiseSet$)
1.[Initialization] **for every** $Var = Val \in PremiseSet$
2. **CREATE-NODE**($Var, Val, [], []$)
3. **repeat**
4. *change* $\leftarrow$ **false**
5. **for each value-inference rule** R_i
6. **for each set of trigger nodes** N_{ij}
7. *change* $\leftarrow$ (**NEW-INFERENCE**(R_i, N_{ij}) **or** *change*)
8. **until** $\neg$*change*

We represent each inference by a node in an inference network. In that network, special nodes are: premise nodes and assumption nodes. A premise node is a variable assignment whose underlying assumption is the empty set. An assumption node is a special node recording an assumption associated with some constraint.

While propagating inferences, statements of the form *node(Var, Val, Support, Dependency)*, are asserted, where Var is a variable assigned the value *Val*, *Support* is the set of assumptions underlying the inference. *Dependency* is the set of variables of all nodes in the proof tree for the current inference. Figure 2 depicts a part of the inference network corresponding to example 2 below. For the node $F = 12$, the set of support and dependencies are: $[A1, M1, M2]$ and $[A, B, C, D, X, Y]$, respectively.

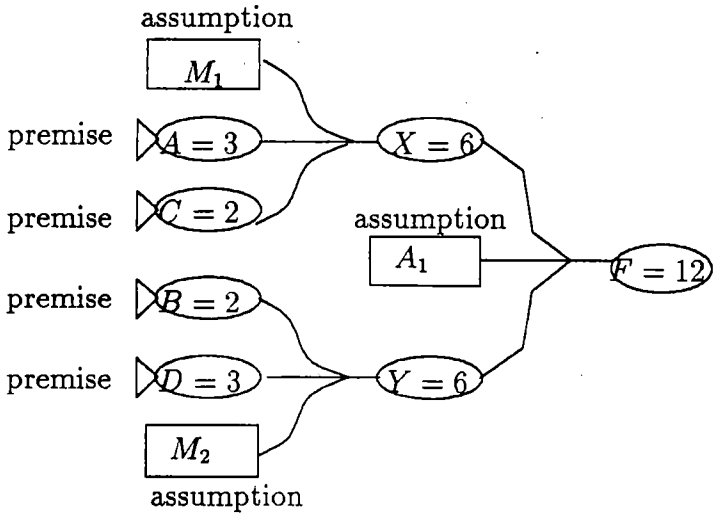


Figure 2: Inference Network

The procedure NEW-INFERENCE called by PROPAGATE does the following. It checks whether the trigger variables of *Nodes* have any dependency that include the conclusion variables of *Rule*; line 5. This check will exclude any circular inference. The support and dependency labels are propagated in lines 8,9, respectively. The conclusion nodes are created in line 11. If a value is deduced for a variable that conflicts with a premise, then the support set for the variable is asserted as a conflict; line 12.

ALGORITHM **NEW-INFERENCE**(*Rule*,*Nodes*,*PremiseSet*)

1. **if** *Rule* **was already triggered by** *Nodes*
2. **then return false**
3. **else begin**
4. $W \leftarrow conclusion\_variables(Constraint, Nodes)$
5. **if for every** $node \in Nodes$ **NO-CYCLE**$(node, W)$
6. **then begin**
7. $H \leftarrow hypothesis(Rule)$
8. $Support \leftarrow \bigcup_{node \in Nodes} support(node) \cup \{H\}$
9. $Dependency \leftarrow \bigcup_{node \in Nodes} (dependency(node) \cup \{node\})$
10. **for every** assignment Val to $Var \in W$
11. CREATE-NODE$(Var, Val, Support, Dependency)$
12. **if** $Var = Val_1 \in PremiseSet$ **and** $Val \neq Val_1$
13. **then assert** *Support* **as a conflict set**
14. **return true**
15. **end**
16. **else return false**
17. **end**

The procedure NO-CYCLE checks that the dependency label for a trigger node does not include any element of the set of conclusion variables W.

ALGORITHM **NO-CYCLE**$(node, W)$

1.**if** $dependency(node) \cap W \neq \emptyset$
2. **then return true**
3. **else return false**

Example 2. *Consider the polybox circuit of figure 3, consisting of three multipliers,* M_1, M_2, M_3, *and two adders. Let the premise set be:* $A = 3$, $B = 2$, $C = 2$, $D = 3$, $E = 3$, $F = 10$, $G = 12$. *Step 1 of PROPAGATE will assert the following premise nodes:*

$node(a, 3, [], [])$ $node(b, 2, [], [])$ $node(c, 2, [], [])$ $node(d, 3, [], [])$
$node(e, 3, [], [])$ $node(f, 10, [], [])$ $node(g, 12, [], [])$

The first cycle of PROPAGATE will assert the following nodes:

$node(x, 6, [m1], [a, c])$
$node(y, 6, [m2], [b, d])$
$node(z, 6, [m3], [c, e])$

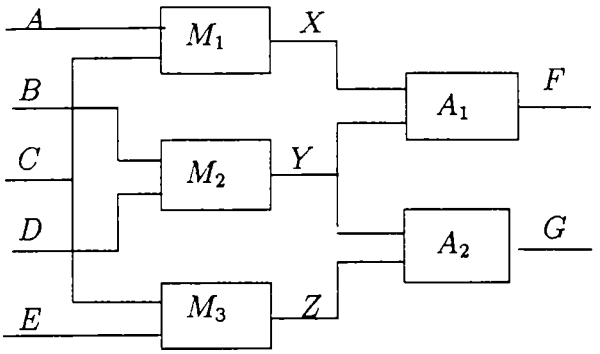


Figure 3: The Polybox Circuit

$node(f, 12, [m2, m1, a1], [b, d, a, c, x, y])$
$node(y, 4, [m1, a1], [a, c, x, f])$
$node(x, 4, [m2, a1], [b, d, y, f])$
$node(g, 12, [m3, m2, a2], [c, e, b, d, y, z])$
$node(g, 10, [m3, m1, a1, a2], [e, a, c, x, f, y, z])$
$node(z, 6, [m2, a2], [b, d, y, g])$
$node(z, 8, [m1, a1, a2], [a, c, x, f, y, g])$
$node(y, 6, [m3, a2], [c, e, z, g])$

The second cycle of PROPAGATE will assert the following nodes:

$node(f, 12, [m3, a2, m1, a1], [e, z, g, a, c, x, y])$
$node(x, 4, [m3, a2, a1], [c, e, z, g, y, f])$

The third cycle cannot deduce anything new, so PROPAGATE stops.

There are two conflict sets: $[m2, m1, a1]$, and $[m3, m1, a1, a2]$.

5 Dynamic Compilation

The procedures presented in section 4 have to be applied every time the premise set has changed. By compiled value inference, we mean generalizing the inferences being made during the propagation process. That generalization permits the inference network to be utilized for various instantiations of the premise set.

This idea of compilation is based on the Explanation-Based Learning (EBL) framework. When given a premise set and constraint propagation is performed, the successful inference rules are also propagated and represented in the inference network. A node in that network consist of an uninstantiated assignment, along with a function relation instantiating the assignment in terms of the instantiation of the premise nodes.

As before, we represent each inference by a node in an inference network. In that network, special nodes are: premise nodes and assumption nodes. A premise node is an uninstantiated (symbolic) assignment to a variable with an empty assumption set. An assumption node is a special node in the network.

While propagating inferences, statements of the form

$$node(Var, SVal, Constraint, Support, Bindings, Dependency) \quad (7)$$

are asserted, where Var is a variable assigned the uninstantiated (symbolic) value $SVal$, $Constraint$ is a function relation expressing $SVal$ in terms of the symbolic values of the premise variables specified by $Bindings$. As before, $Support, Dependency$ denote the set of support and dependencies, respectively.

We call the new propagation process EBL-PROPAGATE, to emphasize the fact that generalized inference network is being learned during the propagation process. Figure 4 depicts the generalized network obtained by EBL-PROPAGATE for the example of figure 2.

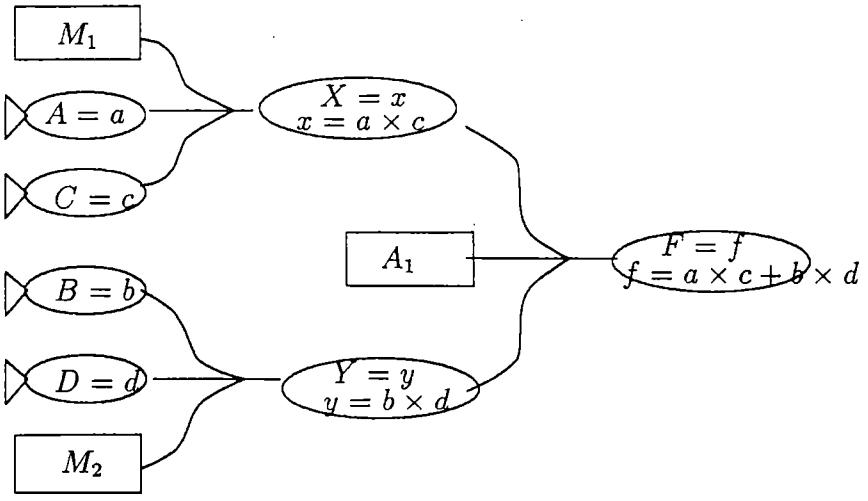


Figure 4: Generalized Inference Netwok

EBL-PROPAGATE generalizes the premise set by replacing each value assignment by a symbolic value, namely a Prolog variable; line 2. It continues applying EBL-NEW-INFERENCE until nothing more can be inferred.

ALGORITHM **EBL-PROPAGATE**($PremiseSet$)

1. [Initialization] **for every** $Var \in variables(PremiseSet)$
2. **CREATE-NODE**($Var, SVal, SVal, nil, [Var : SVal], nil$)
3. **repeat**

4. *change* ← **false**
5. **for each value-inference rule** R_i
6. **for each set of trigger nodes** N_{ij}
7. *change* ← (**EBL-NEW-INFERENCE**($R_i, N_{ij}, PremiseSet$) **or** *change*)
8. **until** $\neg change$

EBL-NEW-INFERENCE first checks whether the inference had been already made; line 1, and if not whether the conclusion variables are included in the dependency lists of any trigger variable (to prevent circular inference); line 5. If the check succeeds then the support, dependency, and binding lists are propagated; lines 7-10. Moreover, the constraints of *Rule* and those of *Nodes* are merged— unifying symbolic values in the process— to obtain a *NewConstraint* list and symbolic values for the conclusion variables. Nodes for the conclusion variables are then created; line 14. If the value of a conclusion variable conflicts with a premise then assert a conflict rule. The rule states that the support set of the conclusion variable is a conflict if its constraints- and bindings-list yield a value for the variable different from a premise assignment.

ALGORITHM **EBL-NEW-INFERENCE**($Rule, Nodes, PremiseSet$)

1. **if** *Rule* **was already triggered by** *Nodes*
2. **then return false**
3. **else begin**
4. $W \leftarrow conclusion\_variables(Rule, Nodes)$
5. **if for every** $node \in Nodes$ NO-CYCLE($node, W$)
6. **then begin**
7. $Bindings \leftarrow \bigcup_{node \in Nodes} bindings(node)$
   /* MERGE-CONSTRAINTS has the side effect of symbolic assignments S-ASSIGN(.) on W */
8. $NewConstraint \leftarrow$ MERGE-CONSTRAINTS($constraints(Rule)$, $\bigcup_{node \in Nodes} constraints(node)$)
9. **if** *NewConstraint* subject to *Bindings* is satisfiable by *PremiseSet* for the the assignment ASSIGN(.) on W
10. **then begin**
11. $H \leftarrow hypothesis(Rule)$
12. $Support \leftarrow \bigcup_{node \in Nodes} support(node) \cup \{H\}$
13. $Dependency \leftarrow \bigcup_{node \in Nodes} dependency(node) \cup \{node\}$
14. **for every** $Var \in W$
15. **begin**
16. $SVal \leftarrow$ S-ASSIGN(Var)
17. $Val \leftarrow$ ASSIGN(Var)
18. CREATE-NODE($Var, SVal, NewConstraint$, $Support, Bindings, Dependency$)
19. **if** $Var = Val_1 \in PremiseSet$ **and** $Val \neq Val_1$

```
20.            then assert a conflict set rule
21.          end
22.          return true
23.        end
24.        else return false
25.      end
26.      else return false
27.  end
```

Example 4. *Consider the polybox circuit of figure 3, with the premise set as in example 2.*

Step 1 of EBL-PROPAGATE will assert the following premise nodes:

$node(a, \_a, [], [], [a : \_a], [])$ $node(b, \_b, [], [], [b : \_b], [])$
$node(c, \_c, [], [], [c : \_c], [])$ $node(d, \_d, [], [], [d : \_d], [])$
$node(e, \_e, [], [], [e : \_e], [])$ $node(f, \_f, [], [], [f : \_f], [])$
$node(g, \_g, [], [], [g : \_g], [])$

The first cycle of EBL-PROPAGATE will assert the following nodes:

$node(x, \_a * \_c, [m1], [], [a : \_a, c : \_c], [a, c])$
$node(y, \_b * \_d, [m2], [], [b : \_b, d : \_d], [b, d])$
$node(z, \_c * \_e, [m3], [], [c : \_c, e : \_e], [c, e])$
$node(f, \_a * \_c + \_b * \_d, [a1, m1, m2], [], [a : \_a, c : \_c, b : \_b, d : \_d], [x, a, c, y, b, d])$
$node(y, \_f - \_a * \_c, [a1, m1], [], [a : \_a, c : \_c, f : \_f], [x, a, c, f])$
$node(x, \_f - \_b * \_d, [a1, m2], [], [b : \_b, d : \_d, f : \_f], [y, b, d, f])$
$node(g, \_b * \_d + \_c * \_e, [a2, m2, m3], [], [b : \_b, d : \_d, c : \_c, e : \_e], [y, b, d, z, c, e])$
$node(g, \_f - \_a\_c + \_c * \_e, [a2, a1, m1, m3], [], [a : \_a, c : \_c, f : \_f, e : \_e], [y, x, a, f, z, c, e])$
$node(z, \_g - \_b * \_d, [a2, m2], [], [b : \_b, d : \_d, g : \_g], [y, b, d, g])$
$node(z, \_g - (\_f - \_a * \_c), [a2, a1, m1], [], [a : \_a, c : \_c, f : \_f, g : \_g], [y, x, a, c, f, g])$
$node(y, \_g - \_c * \_e, [a2, m3], [], [c : \_c, e : \_e, g : \_g], [z, c, e, g])$

The second cycle of EBL-PROPAGATE will assert the following nodes:

$node(f, \_a*\_c+(\_g-\_c*\_e), [a1, m1, a2, m3], [], [a : \_a, c : \_c, e : \_e, g : \_g], [x, a, y, z, c, e, g])$
$node(x, \_f - (\_g - \_c * \_e), [a1, a2, m3], [], [c : \_c, e : \_e, g : \_g, f : \_f], [y, z, c, e, g, f])$

The third cycle cannot deduce anything new, so EBL-PROPAGATE stops. Notice that in this example the symbolic value of each variable is given directly as a function of the premise variable bindings, and therefore the *Constraint* list (eqn. 7) is empty. In general, the value of a variable will be specified as a function relation given by the *Constraint* argument of the node, in terms of the bindings of the premise variables given by the *Bindings* argument (eqn. 7).

In order to better understand how EBL-NEW-INFERENCE obtains the above results, consider applying the value-inference rule (eqn. 6) for the adder $A1$ of the polybox (figure 3). Let the adder's inputs x and y be the trigger variables corresponding to the nodes:

node(x,_a*_c,[m1],[],[a:_a,c:_c],[a,c])

node(y,_b*_d,[m2],[],[b:_b,d:_d],[b,d])

The assumption that a component is not abnormal is simply indicated by a propositional symbol corresponding to the component's label. Propagating the binding sets for x and y (step 7) yields the binding set [a:_a,c:_c,b:_b,d:_d]. The symbolic value for x is $\_a * \_c$. The symbolic value for y is $\_b * \_d$. Unifying the assignments of the adder's inputs with $\_a * \_c$ and $\_b * \_d$ yields $constraint(Rule)$ being equivalent to $SOut = \_a * \_c + \_b * \_d$. Merging constraints (step 8) results in $NewConstraint = \{\}$ and the side effect of a symbolic binding of the adder's output to $SOut = \_a * \_c + \_b * \_d$. The **if** condition of step 9 succeeds with the the value 12 assigned to f. Next the rule's assumption is $a1$. Propagating the support sets for x and y (step 12) yields the support set [a1,m1,m2]. Propagating the dependency sets for x and y(step 13) yields the dependency set [a,c,b,d,x,y]. A node is then created (step 18) for f with the so-obtained symbolic binding, support, dependency, and bindings lists. Since the value of 12 assigned to f conflicts with the premise $f = 10$, the following conflict set rule is asserted (step 19):

$$conflict\_set([m2, m1, a1]) :-$$
$$premise([a : \_a, c : \_c, b : \_b, d : \_d, f : \_f]), diff(\_f, \_a * \_c + \_b * \_d)$$

The rule says given the premise set bindings, if the value of f is different from the value of a times the value of c plus the value of b times the value of d then the set of assumptions [a1, m1, m2] is a conflict (nogood).

Another conflict set rule,

$$conflict\_set([a2, a1, m1, m3]) :-$$
$$premise([a : \_a, f : \_f, c : \_c, e : \_e, g : \_g]), diff(\_g, \_f - \_a * \_c + \_c * \_e)$$

will be asserted following the inference:

$$node(g, \_f - \_a\_c + \_c * \_e, [a2, a1, m1, m3], [], [a : \_a, c : \_c, f : \_f, e : \_e], [y, x, a, f, z, c, e])$$

6 Static Compilation

Static compilation is a pre-compiled value inference, covering all possible assignments of the premise set. We assign symbolic values to the premise variables and propagate the constraints in symbolic form.

The fact that in static compilation the premise variables are uninstantiated leads to the following distinctions from dynamic compilation:

1. Propagated constraints are not evaluated for a specific premise set, since the premise set consists of uninstantiated assignments. This raises the need to ensure that the propagated constraints are compatible.

2. Conflict set rules are formed by hypothesizing all possible inconsistencies between the premise assignments. Note that in dynamic compilation, conflict set rules are only learned when an inconsistency is found for the initially given premise assignments.

This implies that static compilation will definitely lead to a potentially larger set of conflict rules than for dynamic compilation.

Static compilation is performed by the procedure STATIC-PROPAGATE, which takes a set of premise variables and continues propagating the symbolic assignments to those variables until nothing new can be inferred.

ALGORITHM **STATIC-PROPAGATE**($PremiseVars$)
1. [Initialization] **for every** $Var \in PremiseVars$
2. CREATE-NODE($Var, SVal, SVal, nil, [Var : SVal], nil$)
3. **repeat**
4. $change \leftarrow$ **false**
5. **for each value-inference rule** R_i
6. **for each set of trigger nodes** N_{ij}
7. $change \leftarrow$ (**STATIC-NEW-INFERENCE**($R_i, N_{ij}, PremiseVars$) **or** $change$)
8. **until** $\neg change$

STATIC-NEW-INFERENCE is similar to EBL-NEW-INFERENCE with the following two distinctions.

1. The satisfiability check for the merged constraints in step 9 is done with respect to all possible instantiations of the premise set, rather than to an initially given one as in EBL-NEW-INFERENCE.

2. Hypothesized conflict rules are sought for each premise variable, step 18.

ALGORITHM **STATIC-NEW-INFERENCE**($Rule, Nodes, PremiseVars$)
1. **if** $Rule$ **was already triggered by** $Nodes$
2. **then return false**
3. **else begin**
4. $W \leftarrow conclusion\_variables(Rule, Nodes)$
5. **if for every** $node \in Nodes$ NO-CYCLE($node, W$)
6. **then begin**
7. $Bindings \leftarrow \bigcup_{node \in Nodes} bindings(node)$
/* MERGE-CONSTRAINTS has the side effect of symbolic assignments S-ASSIGN(.) on W */
8. $NewConstraint \leftarrow$ MERGE-CONSTRAINTS($constraints(Rule)$, $\bigcup_{node \in Nodes} constraints(node)$)
9. **if** $NewConstraint$ subject to $Bindings$ is satisfiable
10. **then begin**
11. $H \leftarrow hypothesis(Rule)$
12. $Support \leftarrow \bigcup_{node \in Nodes} support(node) \cup \{H\}$

```
13.          Dependency ← ⋃_{node∈Nodes} dependency(node) ∪ {node}
14.          for every Var ∈ W
15.          begin
16.            SVal ← S-ASSIGN(Var)
17.            CREATE-NODE(Var, SVal, NewConstraint,
                       Support, Bindings, Dependency)
18.            if Var ∈ PremiseVars
19.            then assert a conflict set rule
20.            return true
21.          end
22.          return true
23.        end
24.        else return false
25.      end
26.      else return false
27.    end
```

Example 5. *STATIC-PROPAGATE will lead to the same network of nodes as in example 4, but will add a third conflict rule:*

$$conflict\_set([m3, m2, a2], 2) :-$$
$$premise([b : \_b, d : \_d, c : \_c, e : \_e, g : \_g]), diff(\_g, \_b * \_d + \_c * \_e)$$

7 Search Control

Compiling rules for the conflict sets can dramatically reduce the size of the search space. Consider the problem of multiple-fault model-based diagnosis [5]. Given a device consisting of n components, each behaving normally according to a given constraint, and a set of input-output observations for the device (premise set), it is required to find all minimal candidates (multiple as well as single faults) that explain the observations. In other words, it is required to assign credit or blame to components based on observations inconsistency with predictions based on the internal constraints.

The space of potential candidates is potentially exponential in the number of components. The number of conflict sets is generally much lower than the number of potential candidates. How much lower will depend on the connectivity between the components. In one extreme, if we have $2n$ components as in figure 5, we have n conflicts, one for each pair of components $2i$ and $2i+1$, resulting in 2^n candidates. As the connectivity between the components increases, there is the possibility that the number of conflicts may become exponential in the number of components [14].

In cases where the number of conflict sets is a polynomial in the number of components, we expect that static compilation will lead to a considerable reduction in the search space.

The cost of matching the conflict set rules will be less than the cost of propagating the constraints while exploring minimum environments first in the candidate space, as is done in the ATMS-based GDE system [5].

In the case of dynamic compilation, the system learns the conflict set rules as it solves actual diagnostic problems. Since the system is never sure that it has seen enough cases that cover the whole conflict space, the system verifies its candidate hypotheses using constraint-suspension checking. This adds a computational cost that may overwhelm the savings obtained from the compilation.

Computational experiments were carried out using a batch of 100 diagnosis problems, that were created by randomly inducing single to triple faults in the polybox circuit. We fed the same problem batch to three diagnosis systems; first without compilation (MBD), second with dynamic compilation (EBL), and the third with static compilation (STATIC). All three systems produced exactly the same output (all diagnoses) for the same problem, but took different cpu times. We plotted the cumulative cpu time for each diagnosis system for each circuit example. See figures 6.

In EBL (dynamic compilation), the net effect of speed-up from learning conflict set rules, on one hand, and the constraint-suspension checking slow-down on the other hand, depends on the size and the nature of the circuit. For the polybox circuit EBL contributed a net marginal speed-up.

For STATIC (static compilation), conflict sets were generated on the basis of the predetermined conflict rules. The circuit model was no longer needed. The effect of speed-up was very significant, as is evident from the performance plots of figures 6.

8 Conclusions

This paper describes two approaches to compilation in constraint-based reasoning. The first approach, called "dynamic", generalizes and caches truth maintenance labels as infer-

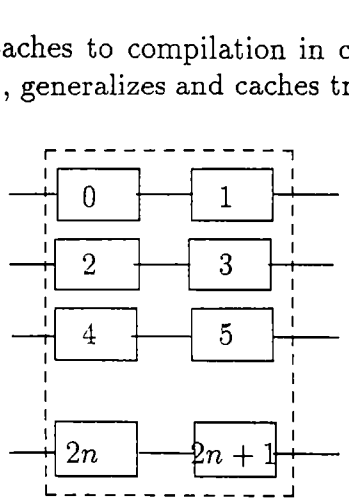


Figure 5: Loosely connected Structure

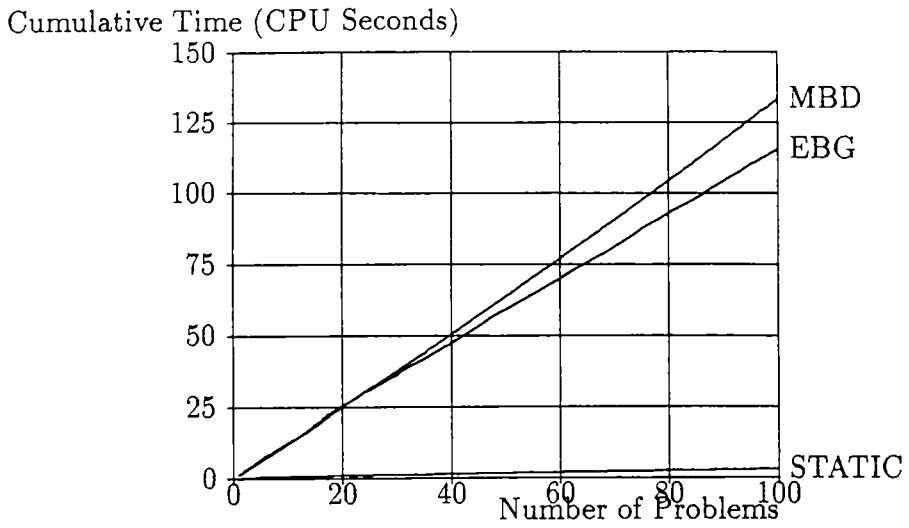


Figure 6: Polybox Empirical Results

ences and queries being made. The second approach, called "static", runs all inferences, generalizes, and caches all labels prior to any queries. In both cases, the labels are stored as rules whose conditions are in terms of premise variable assignments, and conclusion are the inference supporting environments.

Dynamic compilation can be thought of as explanation-based learning, where the generalizations are being made at the time of problem-solving examples. Static compilation corresponds to learning in advance by analyzing all abstract constraints.

The impact of compilation is discussed in terms of a model-based diagnosis application. For multiple-fault diagnosis, compilation is most attractive when the number of conflict sets is orders of magnitude less than the size of the candidate space.

References

[1] J. Cohen. Constraint logic programming languages. *Comm. ACM*, 33:52–68, 1990.

[2] R. Davis. Diagnostic reasoning based on structure and behavior. *Artificial Intelligence*, 24:347–410, 1984.

[3] J. de Kleer. An assumption-based TMS. *Artificial Intelligence*, 28:127–162, 1986.

[4] J. de Kleer. A general labeling algorithm for assumption-based truth maintenance. In *AAAI-88*, pages 188–192, 1988.

[5] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32:97–130, 1987.

[6] R. Dechter. Learning while searching in constraint-satisfaction problems. In *Proceedings, AAAI-86*, pages 178–183, 1986.

[7] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[8] G. DeJong. An introduction to explanation-based learning. In H. E. Shrobe, editor, *Exploring Artificial Intelligence*, chapter 2, pages 45–81. Morgan Kaufmann, 1988.

[9] Y. El Fattah and P. O'Rorke. Learning multiple fault diagnosis. In *Proceedings, CAIA-91*, 1991.

[10] E.C. Freuder. Synthesizing constraint expressions. *Comm. ACM*, 21:958–966, 1978.

[11] P.Van Hentenryck. *Constraint Satisfaction in Logic Programming.* The MIT Press, Cambridge, Massachussetts, 1989.

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

[13] D. McAllester. Truth maintenance. In *Proceedings, AAAI-90*, pages 1109–1116, 1990.

[14] P.Resnick. Generalizing on multiple grounds: Performance learning in model-based troubleshooting. Technical Report AI-TR 1052, MIT Artificial Intelligence Laboratory, February 1989.

[15] G.M. Provan. A complexity analysis of assumption-based truth maintenance systems. In B. Smith and G. Kelleher, editors, *Reason Maintenance Systems and Their Applications*, pages 98–113. J. Wiley& Sons— Ellis Horwood Ltd., New York, 1988.

[16] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[17] R. Reiter and J. de Kleer. Foundations of assumption-based truth maintenance systems. In *Proceedings, AAAI-87*, pages 183–188, 1987.

[18] Group Léa Sombé. Special issue: Reasoning under incomplete information in artificial intelligence. *Int. J. of Intelligent Systems*, 5(4):324–472, 1990.

[19] G.J. Sussman and G.L. Steele Jr. CONSTRAINTS—a language for expressing almost-hierarchical descriptions. *Artificial Intelligence*, 14:1–39, 1980.

[20] S.T. Kedar-Cabelli T.M. Mitchell, R.M. Keller. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

Geometric Constraint Satisfaction Problems

Glenn Kramer, Jahir Pabon, Walid Keirouz, and Robert Young

Schlumberger Laboratory for Computer Science
P.O. Box 200015
Austin, Texas 78720-0015

Abstract

The general constraint satisfaction problem is known to be NP-complete. However, certain domain-specific constraint satisfaction problems can be shown to have polynomial complexity. In this paper, we show that polynomial-time algorithms are possible for solving geometric constraint satisfaction problems (GCSP's). Traditionally, such problems are solved by reformulating the constraints as equations whose roots are found symbolically or numerically. Symbolic solution has exponential complexity, while numerical solution can be unstable and may have robustness problems. Our philosophy for solving GCSP's is to reason directly in the domain of symbolic geometry. We employ an operational semantics for geometric constraint satisfaction, using geometric constructions to measure properties of a model, and actions to move objects in the model to satisfy new constraints without violating previously-satisfied constraints. This incremental approach leads to a monotonic decrease in the number of degrees of freedom in a system of geometric objects, and is responsible for the polynomial complexity of the resulting algorithms. We describe two implemented systems, one for mechanical modeling, and one for kinematic simulation.

1 Introduction

Solving geometric constraint systems is an important problem with applications in many domains, for example: describing mechanical assemblies, constraint-based sketching and design, geometric modeling for CAD, and kinematic analysis of robots and other mechanisms. An important class of such problems involves finding the positions, orientations, and dimensions of a set of geometric entities that satisfy a set of geometric constraints. This paper first examines traditional means of solving geometric constraint satisfaction problems (GCSP's). Then, we introduce a fundamentally different philosophy of constraint satisfaction, based on symbolic geometric reasoning and an operational semantics for constraint satisfaction.

We illustrate this approach in the context of two implemented programs, one for mechanical modeling, and one for kinematic simulation.

Representations of constraint problems must be appropriate for the domain in the sense that the ontology of the problem should lend itself to an efficient solution strategy. A simple translation into the 'standard model' of constraint satisfaction (*e.g.*, [Mackworth, 1977]) may lead to an inefficient, and perhaps exponential-time, solution process, while representation shifts may yield a fast and elegant solution. A classic example is found in the Missionaries and Cannibals problem [Amarel, 1968].

Besides efficiency, many other issues arise in the solution of GCSP's. Underconstrained situations must be dealt with in an intuitive manner, and topological consistency may be important. In domains such as mechanism simulation and mechanical CAD (MCAD), solutions must be consistent in terms of which 'branch' in the solution space is chosen in fully-constrained cases where more than one solution is possible.

1.1 Terminology

The objects of interest in solving GCSP's are called geometric *entities,* or geometric *objects*; some examples are lines, circles, and rigid bodies. Entities have degrees of freedom, which allow them to vary in location or size. For example, in 3D space, a general rigid body has three translational and three rotational degrees of freedom. A circle with a variable radius has three translational, two rotational, and one dimensional degree of freedom (a third rotational degree of freedom is not required because the circle is invariant under the rotation about its axis).

The *configuration variables* of a geometric object are defined as the minimal number of real-valued parameters required to completely specify the object in space. The configuration variables are used to parameterize an object's translational, rotational, and dimensional degrees of freedom (DOF's), with one variable required for each DOF. A *configuration* of an object is a particular assignment of the configuration variables, yielding a unique instantiation of the geometric entity.

The definition of a GCSP is then as follows: Given a set of geometric entities and constraints between them, find the values of the configuration variables of the objects such that all constraints are satisfied. The collection of entities and constraints is called the *constraint system*, or simply the *system.*

2 Equational techniques for solving GCSP's

GCSP's are usually solved by modeling the geometry and constraints with algebraic equations that relate the configuration variables of the different objects according to the problem constraints. Solving these equations—either numerically or symbolically—yields the desired configuration for each geometric entity.

2.1 Numerical solution

Numerical solutions represent constraints using *error terms*, which vanish when the constraint is satisfied, and otherwise have magnitude proportional to the degree to which the constraint is violated. The *error function* is the sum of all error terms; the constraint system is satisfied when the error function is zero. One of the most efficient methods for finding a zero of the error function is Newton-Raphson iteration [Press *et al.*, 1986].

Numerical techniques find zeros of the error function by 'sliding' down the function's gradient. This process is necessarily iterative for nonlinear problems. Numerical techniques have many drawbacks. Each iteration of Newton-Raphson is slow, taking $O(c^3)$ time, where c is the number of constraints. In addition, the Jacobian matrix must be evaluated at every iteration. Overconstrained situations, which are quite common, require pre- and post-analysis to remove redundant constraints before solving and to check them later for consistency. Newton-Raphson can jump chaotically between different roots of the error function during solution [Peitgen and Richter, 1986], which can make the choice of initial solution guess crucially important. Underconstrained situations require pseudo-inverse techniques, since the constraint matrix is non-square. Additionally, when a solution is impossible, no information is available to pinpoint the smallest set of constraints which are inconsistent.

2.2 Symbolic solution

Symbolic solutions use algebraic rewrite rules or other techniques to isolate the configuration variables in the equations in a predominantly serial fashion [Buchberger *et al.*, 1983]. Once a solution is found, it may be reused—or *executed*—on topologically equivalent problems. Execution is fast, typically linear in the number of constraints. If numerical stability is properly addressed, the solution can be more accurate by virtue of being analytic; there is no convergence tolerance as found in numerical techniques. The principal disadvantage of symbolic techniques is the excessive (potentially exponential) time required to find a solution or determine that one does not exist [Liu and Popplestone, 1990]. Poorly-chosen configuration variable assignments can exacerbate the problem by coupling the equations in

unnecessarily complicated ways, requiring very clever and complex inferences. Hence, the symbolic techniques are feasible and complete only for very small problems.

3 Geometric techniques for solving GCSP's

Our approach to solving GCSP's relies on a representation shift from reasoning about configuration variables to reasoning about the DOF's of the actual geometric entities. Configuration variables are related to each other by sets of equations that may be very complicated, tightly coupled, and highly nonlinear; in addition, the domains of the configuration variables are continuous, yielding an infinite search space. In contrast, the degrees of freedom of an object form a compact, discrete-valued, linear description of the state of the object. Coupling of degrees of freedom is rarely encountered, and when it does occur, it can be accommodated easily.

Degrees of freedom form abstract equivalence classes describing the state of a geometric entity without specifying how the constraints that lead to that state are satisfied. DOF's are grouped into three equivalence classes: rotational, translational, and dimensional. All DOF's of the same type are considered identical elements of that resource. DOF resources are consumed by moving an object so as to satisfy a constraint. Further actions are then confined to those that do not violate any previously-satisfied constraints. Therefore, every constraint, upon being satisfied, introduces invariant quantities for the satisfaction of subsequent constraints, and restricts some number of degrees of freedom.

Measurements and actions form the basis for an operational semantics for constraint satisfaction. For example, consider points A and B on a line L, where L has no constraints applied to it. Suppose a constraint specifies that point A be coincident with a fixed point C. The line may be translated to make those two points coincident. If another constraint, say one involving point B on the line, is solved next, any action applied to line L must preserve the location of point A. Therefore, subsequent actions are limited to rotations and scaling about point A. The constraint **coincident**(A, C) removes the line's translational DOF's, thereby restricting subsequent operations. A similar operational semantics is found in [Wang, in preparation].

Reasoning about degrees of freedom is essential to decoupling the constraints. Consider the xyz coordinate frame in Figure 1, with points O, at the origin, and P, in some arbitrary location, rigidly fixed in the coordinate frame. The coordinate frame is parameterized by six configuration variables, three for the translational DOF's, and three for the rotational DOF's. Thus, the coordinate frame is free to translate and rotate in space.

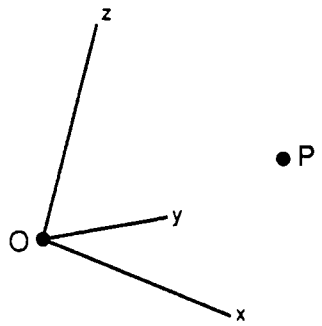


Figure 1: A rigid body with two embedded points.

Fixing the position of either point O or P (through the satisfaction of some constraint) removes the three translational DOF's in the system: the coordinate frame may only rotate about the fixed point in order to satisfy subsequent constraints. But consider the constraints in terms of configuration variables. Fixing the position of point O uniquely determines the three translational configuration variables, while fixing the position of P introduces nonlinear constraint equations into the system to relate the configuration variables to the distance $\overline{OP}$. Solving constraint systems in the configuration variable space is difficult because of this type of coupling between configuration variables. Solving in DOF space is simpler because the actions can be specified independently of how the system is parameterized in terms of configuration variables.

The use of the metaphors of measurement and action to guide equation solution distinguishes our approach from other techniques for solving large sets of nonlinear equations. Since the DOF representation is decoupled, a monotonic decrease in the degrees of freedom in a system can be achieved as the constraints are incrementally satisfied, leading to polynomial-time algorithms for constraint satisfaction.[1]

4 Prototype systems

We believe it is important to examine constraint problems in specific domains, and then generalize our results to the solution of broader classes of GCSP's. To this end, we have

[1]It should be noted that the plan of measurements and actions that satisfy the constraint network do not necessarily correspond to a physically-realizable plan for assembling a collection of real objects. Since the objects in a GCSP are purely geometric, they have no volume or other physical properties. Objects may pass through each other in a ghost-like fashion, on their way to satisfying constraints. This property of the solution process allows decoupling the solution of all constraints affecting any one entity.

implemented our constraint solution ideas in two prototype systems. This section outlines the systems, called HyperGEM and TLA, and then briefly compares them. The HyperGEM program is oriented toward mechanical modeling, and is described in [Keirouz *et al.*, 1990]. TLA is specialized for the domain of kinematic simulation of mechanical linkages, and is described in [Kramer, 1990b].

4.1 HyperGEM

HyperGEM is a prototype environment for mechanical modeling, emphasizing the early stages of 'conceptual' design not addressed by most current MCAD systems. Mechanical modeling includes not only geometric information, but engineering equations and other design information as well. HyperGEM uses a graph-based solution technique for solving GCSP's as a subproblem of conceptual design.

HyperGEM treats the GCSP as a graph flow problem, where the constraints are sources of DOF's, and the geometric entities are sinks. A graph algorithm allocates the dimensional, translational, and rotational DOF's absorbed by each entity so as to distribute the DOF equivalence classes according to each entity's ability to absorb them. The algorithm is executed incrementally each time a new constraint is added to the system. When all DOF's have been allocated, an ordered dependency list among the geometric entities is generated. This dependency list can be used as an execution sequence to update the geometric entities so as to satisfy the imposed constraints. The execution sequence is recomputed after a new constraint is added, as it may alter the allocation of DOF's.

The DOF's are represented in a hierarchical fashion from 'weakest' to 'strongest': dimensional, translational and rotational. An action that restricts a particular class of DOF may be used to satisfy a constraint involving that class of DOF or a weaker one.

Default procedures are provided to ensure 'intuitive' behavior of the constraint solver in underconstrained situations, which are quite common in conceptual design. As the allocation of DOF's is incremental, overconstraining constraints are detected at the time they are added to the system. The solver does not yet attempt to classify the overconstraining conditions (*i.e.*, whether they are conflicting, in which case no solution is possible; or just redundant but solvable); it 'flags' the constraints for the user, and ignores them when updating is required. Cycles in the constraint graph are identified and solved numerically.

4.2 TLA

TLA is a program for kinematic analysis of mechanisms. Since many mechanical devices contain loops, TLA is oriented toward the solution of constraint loops. A graph reduction

algorithm is used in which constraint loops are identified and solved, reducing a cycle in the constraint graph to a single node, and proceeding recursively.

To solve cycles in the constraint graph, it is necessary to have knowledge not only of how actions can satisfy constraints, but also of how partially-constrained objects may move as a function of their available DOF's. The loci of points, lines, and vectors on partially-constrained objects are intersected as a geometric analog to solving simultaneous nonlinear equations. The complete algorithm is called degrees of freedom analysis, and is described in detail in [Kramer, 1990a].

Information about measurements and actions is stored in a dispatch table indexed by translational DOF's, rotational DOF's, and the type of constraint to be solved.[2] Each entry in the table specifies how to move an object to satisfy the new constraint using only available DOF's, and what DOF's the object will have after the action is performed. Similar data structures store information about the loci of points, lines, and vectors on bodies as a function of translational and rotational DOF's.

Kinematic simulation of a mechanism involves repeatedly solving the same set of constraints (with a few numerical parameters, such as the angles of lines or displacements of points, changing for each solution). Therefore the measurements and actions for solving the constraint system is compiled into an efficient procedure for reuse on any mechanism of the same topology.

4.3 Comparisons

The two programs described above appear to have complementary sets of strengths and weaknesses, as briefly described here.

First, TLA cannot represent some situations involving coupled DOF's that occur in MCAD (such situations cannot occur in the kinematics of mechanical linkages); in contrast, the hierarchical representation of DOF's in HyperGEM overcomes this problem. Second, HyperGEM uses iterative techniques to solve some loops that could be solved analytically. The loop analysis capability of TLA allows closed-form analytic solution of such loops. And finally, TLA compiles plans for reuse in simulation, while HyperGEM does not. This is in part because in TLA's domain of kinematics, the constraint set is static (although numerical values of constraint parameters can change), whereas constraints are continually added and removed from HyperGEM's constraint set during the process of conceptual design. We have begun the process of combining the two techniques to obtain a powerful, more general-purpose algorithm for solving GCSP's.

[2] In kinematics, all objects are rigid bodies, so no dimensional DOF's are involved.

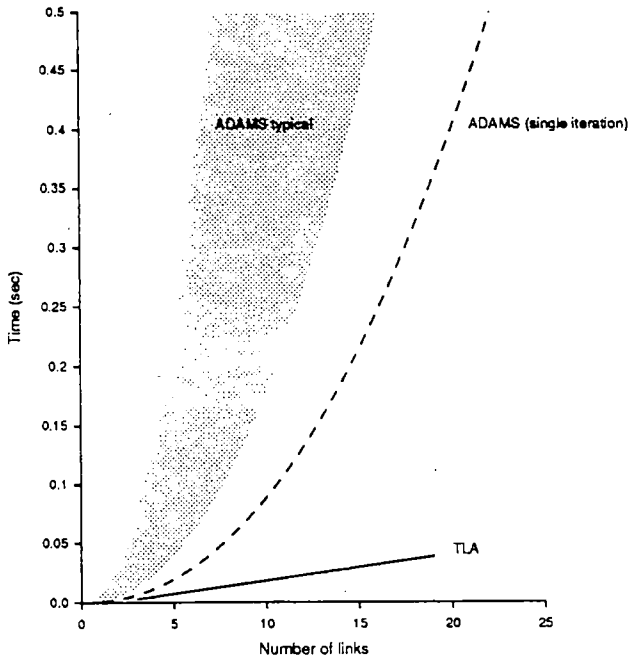


Figure 2: Timing comparisons of TLA and ADAMS.

5 Theoretical and empirical analysis

HyperGEM's graph algorithm for allocating DOF's involves an amount of work linear in the number of constraints each time a new constraint is added to the system. If only numerical parameters change between constraint solutions, and no constraints are added or deleted, the solution can be re-executed in typically linear time (not including numerical solution of constraint loops). When a constraint is added or deleted, the graph flow algorithm is run again.

For TLA, a plan to satisfy a GCSP is generated in $O(gc)$ time, where g is the number of geometric entities in the constraint system, and c is the number of constraints. The plans may be executed in $O(g \log g)$ time, although typically the execution time is linear in g.

TLA has also been empirically compared with the ADAMS mechanism simulator, which employs iterative numerical solution techniques [ADAMS, 1987]. The graph of Figure 2 shows the runtime of ADAMS and TLA as a function of the number of links in a mechanism. The dashed line shows the time per iteration for ADAMS; typically, between 2 and 12 iterations are required to solve a GCSP, as indicated by the gray area. In contrast, the behavior of TLA is linear, and is substantially more efficient.

6 Discussion

The general constraint satisfaction problem was defined by [Mackworth, 1977]. In this restricted sense, a constraint problem involves finding consistent bindings for a set of variables in a logical formula (a conjunction of unary and binary predicates), where each variable may take one of only a finite set of discrete values. More recently, problems with infinite (*i.e.*, continuous) domains have become of interest, as in temporal reasoning [Allen and Hayes, 1985] and spatial reasoning [Popplestone *et al.*, 1980].

It seems unlikely that there will ever be an efficient solution technique to general constraint problems. However, domain-specific strategies can be quite efficient; sometimes they are a complete solution strategy, and other times they serve to accelerate the solution of the vast majority of that domain's constraint satisfaction problems.

In this context, we view our work as a small step toward the solution of the general constraint satisfaction problem. While it provides leverage in solving GCSP's, it is doubtful that the methods will extend beyond the realm of geometry. Just as weak problem-solving methods like GPS [Newell and Simon, 1972] were supplanted by more domain-specific approaches, the general constraint satisfaction paradigm must be supplemented with domain-specific algorithms.

There are some broad concepts that can be reused in formulating constraint satisfaction problems for other domains. The notion of abstracting some continuous space (*e.g.*, position and orientation) into a discrete space (*e.g.*, degrees of freedom) may apply to other domains. Designing algorithms that make use of monotonic trends (such as the reduction of degrees of freedom of a geometric entity) tends to lead to polynomial-time algorithms. Creative representation shifts will be required to use these principles in other domains, but if they can be found, the benefits may be substantial.

References

[ADAMS, 1987] Mechanism Dynamics, Inc., Ann Arbor, Michigan. ADAMS *Users Manual*, 1987.

[Allen and Hayes, 1985] James F. Allen and Patrick J. Hayes. A common-sense theory of time. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 528–531, Los Angeles, California, August 1985.

[Amarel, 1968] Saul Amarel. On representations of problems of reasoning about actions. In D. Michie, editor, *Machine Intelligence 3*, pages 131–171. Edinburgh University Press, Edinburgh, Scotland, 1968.

[Buchberger *et al.*, 1983] B. Buchberger, G. E. Collins, and R. Loos, editors. *Computer Algebra: Symbolic and Algebraic Computation.* Springer-Verlag, Wien, second edition, 1983.

[Keirouz *et al.*, 1990] W. Keirouz, J. Pabon, and R. Young. Integrating parametric geometry, features, and variational modeling for conceptual design. In J.R. Rinderle, editor, *Proc. ASME Design Theory and Methodology Conference*, pages 1–9. American Society of Mechanical Engineers, 1990.

[Kramer, 1990a] Glenn A. Kramer. *Geometric Reasoning in the Kinematic Analysis of Mechanisms.* Dphil thesis, University of Sussex, Brighton, UK, October 1990.

[Kramer, 1990b] Glenn A. Kramer. Solving geometric constraint systems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 708–714, Boston, Massachusetts, July 1990.

[Liu and Popplestone, 1990] Yanxi Liu and Robin J. Popplestone. Symmetry constraint inference in assembly planning: Automatic assembly configuration specification. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1038–1044, Boston, Massachusetts, 1990.

[Mackworth, 1977] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[Newell and Simon, 1972] A. Newell and H. A. Simon. *Human Problem Solving.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

[Peitgen and Richter, 1986] H.-O. Peitgen and P. H. Richter. *The Beauty of Fractals: Images of Complex Dynamical Systems.* Springer-Verlag, Berlin, 1986.

[Popplestone *et al.*, 1980] R. J. Popplestone, A. P. Ambler, and I. M. Bellos. An interpreter for a language for describing assemblies. *Artificial Intelligence*, 14(1):79–107, August 1980.

[Press *et al.*, 1986] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, Cambridge, UK, 1986.

[Wang, in preparation] Bin Wang. *An Operational Approach for Constraint Satisfaction.* PhD thesis, University of Southern California, Los Angeles, California, (in preparation).

Constraint Imperative Programming Languages

Bjorn Freeman-Benson and Alan Borning
University of Washington, Dept. of Computer Science and Eng., FR-35, Seattle, WA, 91895

AAAI Spring Symposium on Constraint-Based Reasoning
draft prepared: November 16, 1990

1 Introduction

A primary motivation for this research is investigating programming language support for interactive graphical applications. Our previous research made it clear that some portions of an interactive system are most naturally described as a set of relations, others as a sequence of operations. *Constraint* programming languages are based on multidirectional relations for specifying the desired results, but in their pure form have no means of specifying how to attain those results. Thus the language's embedded constraint solver is given the task of determining the algorithms to use and in what order they should be applied. *Imperative* languages are based on sequences of operations, and allow an exact and precise specification of the algorithm to use in computing a result. In other words, constraint languages emphasize the *result* whereas imperative languages emphasize the *process*. The core of our work has been to combine these two (apparently incompatible) paradigms in a single framework: Constraint Imperative Programming (CIP). Our thesis is that this integration is reasonable, expressive, and useful.

Constraint Imperative Programming permits programmers to choose whichever is the appropriate paradigm for each part of the interactive system: constraints for relations, and imperative code for sequencing. For example, in a user interface, the output channel can be nicely described using constraints, or filters, that relate internal data objects to graphical displays [Ege et al. 87]. Conversely, the input channel is more conveniently described with state transitions, event handlers, mode sequencing, and other imperative control flow operators. Analogously, within the application, numerous consistency and preference relations are easily expressed as constraints: arithmetic relations between numbers, information about which nodes in a graph are adjacent to each other, and rules about how the font size should relate the number of lines on a page, to name just a few. Other aspects of the application, however, are better specified imperatively.

In our work we extend the notion of constraints to constraint hierarchies, which allow both required and non-required (i.e., preferential or default) constraints [Borning et al. 89a]. A common use of such default constraints is to specify that objects remain in the same state over time, unless there is some reason for them to change. In an interactive graphics application, for example, this means that as we edit parts of a picture, other parts don't change gratuitously. In Constraint Imperative Programming, such defaults are ubiquitous (variables stay the same unless changed as a result of some stronger constraint); they provide a solution to the classical AI Frame Problem in these languages. Constraint hierarchies are also useful for expressing user preferences, for example that *window1* be above *window2* if possible, and (less strongly) that *window1* be on the left-hand-side of the screen. The theory of constraint hierarchies is more fully described in reference [Borning et al. 89a].

2 — The Semantics

The key to integrating the declarative constraint paradigm and the imperative object-oriented one is the definition of a semantics that combines the two. As a starting point, one can loosely characterize the semantics of the imperative and declarative paradigms:

Imperative Each instance variable holds a single value (e.g., a pointer to an object). Also, each instance variable potentially holds a different value after each instruction is executed. However, the value of a variable cannot change unless the instruction explicitly writes to that variable.

Declarative Each variable holds only one value, i.e., the result of evaluating the program. For functional programs, this is the least fixed-point; for constraint hierarchies, the best valuation. Time does not advance, and the value of the variable cannot change[†].

Constraint Imperative Programming (CIP) is a framework for a family of languages that merge declarative constraints with imperative state and sequencing. This framework defines the essential semantics of CIP languages, but leaves language designers free to create or adapt their own syntax. Our initial CIP language was loosely based on Ada and Object Pascal; our current language, Kaleidoscope, is based on Smalltalk [Freeman-Benson 90a].

The complete CIP semantics are described in [Freeman-Benson 90b]. The major points are as follows:

- Each variable holds a stream, or history, of values. Each value represents the value of the variable at a different interval, with subsequent values representing subsequent intervals. Time is virtual and represented by the positive integers. These values are held by sub-variables named *pellucid variables*[‡]. For example, at time n the variable `X` represents the stream of pellucid variables $x_1, x_2, \ldots, x_n$. The pellucid variables for past intervals are $x_1, x_2, \ldots, x_{n-1}$; the pellucid variable for the current interval is x_n; and the pellucid variables for the future do not yet exist ($x_{n+1}, \ldots$). To prevent paradoxes that can arise when past values change, pellucid variables are write-once. Note that pellucid variables are a semantic feature and are not available to the programmer—he or she can only use variables that denote entire streams.

- *Frame axioms* (the term is borrowed from the artificial intelligence literature) state that variables stay the same unless explicitly changed. Imperative programs implicitly satisfy the frame axioms because only assignment can change a variable's value. Pure constraint programs do not have a notion of time and thus do not suffer from the frame problem. In Constraint Imperative Programming, a special very weak *stay* constraint is used to represent the frame axioms: $\forall t$, `very_weak` $V_t = V_{t-1}$

- An imperative assignment statement represents a constraint on the next pellucid variable of the stream and thus can only affect the next interval. It can affect the distant future only when the new value is propagated forward by other constraints (such as the weak stays mentioned in the previous paragraph). For example, if the current value of time is 9, then `x — x + 3` is equivalent to $x_{10} = x_9 + 3$. Thus, as time continues to advance, this assignment constraint will fade into the past.

[†]Logic variables in logic programs can be progressively refined during execution, but they cannot arbitrarily change from one value to another.

[‡]Pellucid: transparent, translucent. Pellucid variables are essential to the semantics of CIP, but cannot be directly accessed by the program. Thus they are "transparent or translucent."

- All other constraint expressions denote a (potentially) infinite set of constraints on individual values in the streams. Thus, in Constraint Imperative Programming, the constraint expression $x = y$ defines a value constraint for each instant starting with the current time: $x_t = y_t$, $x_{t+1} = y_{t+1}, \ldots$ In other words, a constraint expression affects the values of the variables it constrains "from now on". A `while... assert...` construct corresponds to "from now until then," or: $\forall t \in m \ldots n,\ x_t = y_t$

- Virtual time is explicitly advanced using the "#" operator. Note that explicitly separating time advances from other statements allows simultaneous assignments, so that for example one can conveniently swap two variables without using a temporary:

```
x — y;                    begin
y — x; #      ≡             x — y;      ≡      x(t+1) = y(t)
                            y — x;             y(t+1) = x(t)
                          end; #
```

A CIP program without constraints (i.e., only assignments), has similar semantics to an imperative program (values stay the same, assignments change values). A CIP program without assignments or hash-marks has the same semantics as a pure constraint program.

3 — Comparisons with Other Paradigms

3.1 — Versus C++ (Better Aliasing)

One way to understand the Constraint Imperative Programming semantics is to treat CIP as an imperative language with *better aliasing*. In this view, a CIP program is an imperative program in which invisible relations (constraints) can be defined between memory cells. These relations are similar to those created by deliberate aliasing. For example, in the C++ programming language an alias is created when two pointers point to the same object (e.g., `p` and `q` both point to object #124.) When object `*p` is modified, object `*q` is too (because `*p` and `*q` are the same object). In a CIP language, the same aliasing effect can be created by a **`required`** equality constraint. Additionally, CIP languages go beyond mere equality and support fairly arbitrary constraints between memory cells (see figure 1.)

C++	*Constraint Imperative Programming*
`int z, *p, *q;`	`var p, q, r;`
`p = &z;`	`always: p = q;` *% ⇐ alias created*
`q = &z;` *`/* ⇐ alias created */`*	`always: q - 32.0 = r * 1.8;`
	% ⇑ C++ cannot do this

Figure 1: Constraint Imperative Programming as Better Aliasing

3.2 — Versus CLP (Constructionism)

Another way to understand the semantics is to treat CIP as an imperative program that *constructs* a constraint graph. In this view, a CIP virtual machine is composed of two parts which cooperate

	cc languages	*Constraint Imperative Programming*
Constructor	Logic programming	Imperative programming
Backtracking	Yes	No
Constraints	Flat (all required)	Constraint hierarchy
Adding constraints	Tell	Constraint statement
Querying constraints	Ask a constraint	Demand a value
Value refinement	Yes	Yes until commit then no
Adding variables	Tell a constraint with new variables	Both time advances and variable declarations
Objects	No	Yes
Long-lived constraints	Unnecessary	Automatic
Familiar to imperative programmers	No	Yes

Table 1: Comparing the cc languages with Constraint Imperative Programming

to construct and solve a network of constraints. The two parts are an imperative execution engine and an constraint-based data store. The imperative engine interacts with the passive data store by asserting constraints and requesting the values of pellucid variables:

Imperative Engine		*Constraint-based Data Store*
constraint statement	—	add a constraint
constraint statement	—	add a constraint
advance time	—	append pellucid variables
assignment	—	add a constraint
conditional branch	—	solve constraints, commit, and return a value
constraint statement	—	add a constraint
advance time	—	append pellucid variables
... *etc.*...		... *etc.*...

This constructionist view can also be applied to a restricted version of the cc family of languages [Saraswat 89] (see table 1) and to the Constraint Logic Programming and Hierarchical Constraint Logic Programming frameworks [Jaffar & Lassez 87, Borning et al. 89b]. In these logic programming-based constraint languages a logic program, rather than an imperative program, does the constructing and querying of the passive constraint database.

3.3 — Versus UIMSes (Finite State Machines) —

A third way to describe the semantics (for a restricted class of programs) is to view a CIP program as a large *finite state machine*, similar to the way that many User Interface Management Systems operate[†]. The constraint statements define both inter- and intra-state data relations, and the imperative control flow statements define the state transitions. Always constraints define relations that are true for all states; once constraints define relations that are true for just one state; and the "**assert *P* during *B***" construct defines constraints that are true for some subset of the states. Stay constraints and assignments define constraints between pairs of states.

[†]CIP languages are Turing-equivalent, and thus, in general, cannot be modeled by a finite state machine. However, the FSM view is useful in many cases.

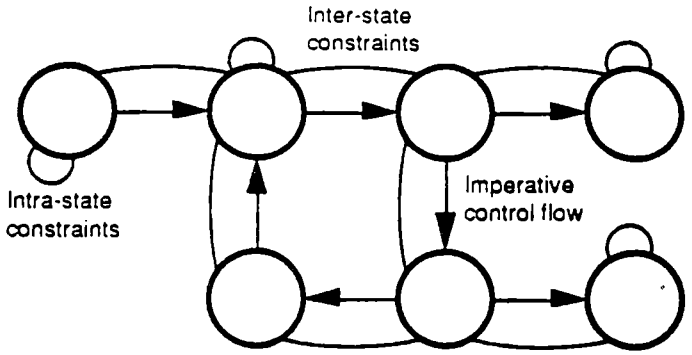


Figure 2: A Constraint Imperative Program as a Finite State Machine

The finite state machine visualization is particularly apt for interactive user interfaces: each state corresponds to one mode (IDLE, DRAGGING-ICON, DRAGGING-ICON-WITH-SHIFT-KEY, ...), and the imperative state transitions correspond to significant events (mouse button down, key pressed, ...). Constraints common to all modes are defined with always expressions, but the constraints unique to each mode are created by once expressions. For example, the menu bar is always at the top of the screen, but only in the DRAGGING-ICON mode is the icon's position equal to the mouse position.

References

[Borning et al. 89a] Alan Borning, Bjorn Freeman-Benson, and Molly Wilson. Constraint hierarchies, November 1989. Submitted to ACM Transactions on Programming Languages and Systems.

[Borning et al. 89b] Alan Borning, Michael Maher, Amy Martindale, and Molly Wilson. Constraint Hierarchies and Logic Programming. In *Proceedings of the Sixth International Logic Programming Conference*, pages 149–164, Lisbon, Portugal, June 1989.

[Ege et al. 87] Raimund K. Ege, David Maier, and Alan Borning. Building Interfaces with Constraints. In *Proceedings of the Second International Conference on Human-Computer Interaction*, 1987.

[Freeman-Benson 90a] Bjorn Freeman-Benson. Kaleidoscope: Mixing Objects, Constraints, and Imperative Programming. In *Proceedings of the 1990 Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*, pages 77–88. ACM SIGPLAN, October 1990.

[Freeman-Benson 90b] Bjorn N. Freeman-Benson. The Evolution of Constraint Imperative Programming. Technical Report 90-05, Laboratoire d'Informatique, Université de Nantes, Nantes, France, August 1990.

[Jaffar & Lassez 87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM Principles of Programming Languages Conference*, Munich, January 1987. ACM.

[Saraswat 89] Vijay Anand Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, Computer Science Department, January 1989.

Explaining Constraint Computations

Leon Sterling & Venkatesh Srinivasan
Department of Computer Engineering and Science &
Center for Automation and Intelligent Systems Research,
Case Western Reserve University,
Cleveland, OH 44106

Abstract

Explanation is regarded as an essential component of AI systems, especially expert systems. Technology for explaining rule-based systems is particularly well-understood. Constraint systems can be viewed as generalizations of rule-systems where there is more flexibility in the direction of information flow. Explaining this more flexible framework is difficult and has not received much attention. In this paper, we propose a model for explaining constraint computations based on traversing *reduction trees*, an analogue of *Prolog proof trees*. Example explanations are given for computations with the constraint logic programming language CLP($\mathcal{R}$).

1 Introduction

It is widely accepted that most intelligent systems need an explanation component. People interacting with a computer demand justification of the computer's behavior in terms that they understand. "The computer told me" is not, and should not be, sufficient basis for decisions.

How to explain rule-based systems is well understood [6]. The rules involved in a computation must be explicitly represented, and collected in some structure such as a proof tree [7]. The explanation is generated from some traversal of the proof tree, with due consideration made of user interface issues.

How to explain constraint computations is not well understood as there are difficulties in directly translating the techniques from rule-based systems. For example, although constraint logic programming languages are natural generalizations of languages such as Prolog, meta-language capabilities are not currently present in them. Also proof trees are inadequate to explain the proof structure for a constraint logic programming computation because the latter generalizes unification by a more general mechanism - solving constraints in the domain of interpreted functors over terms in the domain of interpretation. What meta-level linguistic capabilities are needed is only now beginning to be understood [4,5].

This paper proposes an explanation style for constraint computations. It focuses on explaining the logical component of the constraint computation and treats the constraint solver as a *black box*. The syntactic structure of the constraint program is considered important so that the programmer can determine the explanation given by the program.

Section 2 gives our model of constraint computation which is essentially a general form of the model of constraint logic programming proposed by Lassez and Jaffar in [3]. In Section 3, we propose a new structure, a reduction tree, as a graphical form of representing a constraint computation. *Reduction trees* are analogous to proof trees for Prolog computations. Explanations for constraint computations are then generated by traversing the reduction tree. Sample explanations for CLP($\mathcal{R}$) computations are given in Section 4.

We have developed a prototype program in Prolog which generates explanations from reduction trees given in the examples. Such a program is not currently implementable, as far as we know, in a constraint programming language due to the lack of suitable meta-predicates. Consequently, this paper can implicitly be seen as indicating to developers of constraint programming languages what system meta-predicates need to be provided.

2 Constraint Logic Programming Model

A constraint logic programming language consists of a domain of interpretation, a collection of function symbols denoted by Σ, a collection of variables V, and a collection of predicate symbols Π.

A term is a variable, a constant or is of the form $f(t_1, \ldots, t_n)$ where $f \in \Sigma$ and $t_1, \ldots, t_n$ are (argument) terms. A term s is a subterm of t iff $s \equiv t$, or s is the subterm of an argument of t. We will need to refer to specific subterms, and introduce a suitable numbering scheme. The first subterm t_1 in $f(t_1, \ldots, t_n)$ will be identified by 1 and t_2 by 2 etc. Further nesting of subterms within a particular term is handled by giving an extension to the first number, e.g. in *t=f(g(h(a,b),2),a)* *b* can be identified as 1.1.2 since $g(h(a,b))$ is the first subterm of t, $h(a,b)$ is the first subterm of $g(h(a,b))$ and so forth.

In contrast to Prolog, the function symbols are divided into two classes, interpreted function symbols denoted by Σ_i, and uninterpreted function symbols denoted by Σ_u. Predicate symbols Π are also divided into two classes, constraint symbols denoted by Π_c, and remaining predicate symbols denoted by Π_p. *Constraints* are of the form $p(t_1, \ldots, t_n)$ where p is an n-ary element of Π_c and $t_1, \ldots, t_n$ are terms. A goal is of the form $p(t_1, \ldots, t_n)$ where $p \in \Pi$ and

$t_1, \ldots, t_n$ are terms.

A *constraint logic program* is a finite set of clauses of the form :

$$\mathrm{A} \leftarrow C_1, \ldots, C_m, B_1, ..., B_n$$

where A is a logical atom, $C_1, \ldots, C_m$ are constraints, $B_1, ..., B_n$ are logical atoms and m, n are non-negative integers.

We will occasionally refer to the $C_i's$ as *explicit constraints.*

A *reduction step* of a computation of a constraint logic programming (CLP) language program P selects a goal :

$$(C \square D_1, ...D_i, ...D_o)$$

where C is a satisfiable conjunction of constraints, and each D_i is a logical atom; selects an atom D_i, and chooses a rule of the form $A \leftarrow C_1, \ldots, C_m, B_1, ..., B_n$, such that A and D_i have the same predicate symbol. It then adds constraints arising from equating the terms of A with those of D_i, to C resulting in a *satisfiable set of constraints.* The new resolvent is

$$(\{C \wedge C_1, \ldots, C_m \wedge \{ A = D_i\}\} \square D_1, ..., D_{i-1}, B_1, ...B_n, D_{i+1}, ...D_o)$$

where $C \wedge C_1, \ldots, C_m \wedge \{ A = D_i \}$ are satisfiable with $\{ A = D_i \}$ being a set of constraints arising out of equating the arguments of A and D_i.

It will be useful to identify constraints which equate a variable with a term whose principal functor is an interpreted function symbol. *We refer to such constraints as implicit constraints.*

Operationally, we can think of a computation of a constraint logic program as a sequence of reduction steps - accumulating constraints and either verifying that the constraints are satisfiable, or else backtracking if they are not. The computation terminates with an accumulated set of constraints.

In this paper we give explanations for a particular language, CLP($\mathcal{R}$), which is an example of a constraint logic programming language. The domain of CLP($\mathcal{R}$) is the set of real numbers $\mathcal{R}$, the interpreted function symbols are {+, -, *, /, transcendental functions, pow}, the constraint symbols are {=, <, <, $\geq$, $\leq$}, the user-defined function and predicate symbols are the uninterpreted function and predicate symbols respectively. For a complete definition, please refer to [2].

3 Reduction Trees

Explanations of Prolog computations are based on traversing a proof tree [7]. To explain computations of constraint logic programs, we need a structure analogous to a proof tree. In this section we define a reduction tree, and show how it applies to constraint computations. Throughout we use the word program to

refer to a constraint logic programming language program.

A *reduction tree* for a computation of a program starting from a goal G is a directed tree (V,E) whose root is G. Each vertex V has the form (B,C) where B and C are the following components respectively :

- Goal component
- Constraint component

If the clause A $\leftarrow C_1, \ldots, C_m, B_1, \ldots, B_n$ is used in the computation then there is an edge from a pair (A,C_A) to a node (B',C') where either

(1) B' is *empty*, and C' is one of the $C_i s$.

or

(2) B' is one of the $B_j s$ and C' is a conjunction of implicit constraints (defined earlier) arising from the arguments of B' together with constraints obtained from the accumulated constraints which contain any occurrence of the variable occurring in the arguments of B'.

The definition of reduction trees is consistent with the computation model of constraint programming language presented earlier, and the goal component of every *non-leaf node* in the reduction tree is one of the $B_j s$ chosen at each reduction step. The *explicit constraints* (defined earlier) can appear only as the constraint component of *leaf nodes* of the reduction tree because by the definition given above, no directed arc can originate from one of the $C_i s$ in the body of any clause. Another important difference between a reduction tree and proof tree comes from the fact that proof trees in Prolog do not contain variables while reduction trees may contain variables. This comes about because answers to a CLP computation can be constraints over variables appearing in the goal arguments.

3.1 Goal Component Representation

The goal component of a node in the reduction tree is similar to a node in a Prolog proof tree. Differences between them will be pointed out in this subsection. The differences arise due to the presence of implicit constraints in the goal head. The goal component of a node in the reduction tree is absent for explicit constraints, and is denoted as *empty*. For example, an explicit constraint $X > 10$ with X bound to 11 will appear in the reduction tree as *(empty,{11>10})*.

If any subterm of the head of the clause used in the computation has as principal functor an interpreted function symbol, then there will be an implicit constraint in the constraint component of the node whose goal component was

equated to the head of the clause in the computation. For example, for a goal *f(X,N)* and clause *f(X,N1+N2)* :- ... if $N1$ and $N2$ are 2 and 3 respectively, the reduction tree node for *f(X,N)* would be $f(\_, 5)$ with a corresponding constraint 5=2+3 appearing in the constraint component. If $N1$ and $N2$ are not instantiated, the goal component of the reduction tree node will be $f(\_, T1)$ with $T1 = N1 + N2$ being included in the corresponding constraint component. Note that this corresponds to the introduction of a *temporary variable.*

3.2 Constraint Component Representation

The constraint component of a reduction tree node contains representations of any implicit constraints as well as solved and unsolved constraints over all variables appearing in the head of a clause.

The constraint component is a set of *named* and *miscellaneous* constraints. A named constraint has the form \$i : C where i identifies a subterm of the logical atom in the corresponding goal component and C is an implicit constraint. Miscellaneous constraints appear only if there are unsolved constraints involving variables in the goal component. These are precisely the simplification of the constraints in the *satisfiable constraint set* involving variables appearing in the *logical atom* in the goal component excluding the implicit constraints already mentioned.

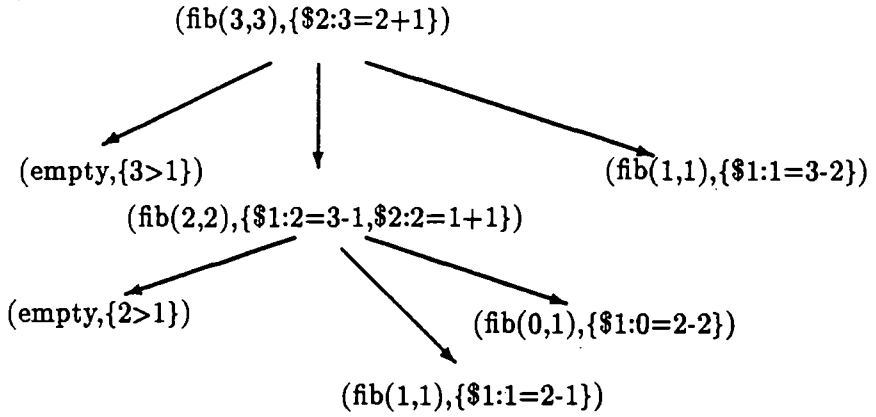
Subterm numbers are given because two different terms in a goal component might have the same values but different constraint component values, e.g. in $g(3, 3)$ the first 3 could be a unified value and the second one could arise from the constraint 3=2+1. Component numbering usage resolves ambiguities and the corresponding constraint component will appear as { \$2 : 3=2+1 }.

Here are reduction trees for some CLP($\mathcal{R}$) programs:

Example 1:

```
fib(0,1).
fib(1,1).
fib(N,X1+X2) :- N > 1, fib(N-1,X1), fib(N-2,X2).
```

The goal ?- fib(3,N) has the solution $N = 3$. The corresponding reduction tree is

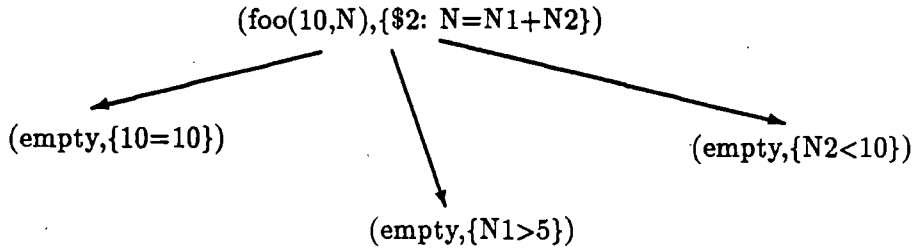


This example, albeit simple illustrates a reduction tree highlighting the constraint computation mechanism for computing fibonacci numbers. Both implicit and explicit constraints appear in the figure.

Example 2:

foo(X,N1+N2) :- X=10, N1> 5, N2 < 10.

The goal ?- foo(10,N) has the solution *true*. The corresponding reduction tree is



The second example illustrates a reduction tree having uninstantiated variables in some of the arguments in the clause head. There is a named implicit constraint in the constraint component. The answer to the query in the CLP($\mathcal{R}$) system is *true* which is not very informative.

The third example, adapted from [1], illustrates a case when constraints appear in the miscellaneous constraint set, and also uninstantiated instances of variables appear in the goal head:
Consider the following predicate for a point X,Y to be on the circumference of a circle with center A,B.

on_circle(p(X,Y),c(A,B,$(A-X)^2+(B-Y)^2$)).

For a query ?- on_circle(p(2,0),C) , the corresponding node in the reduction tree would be

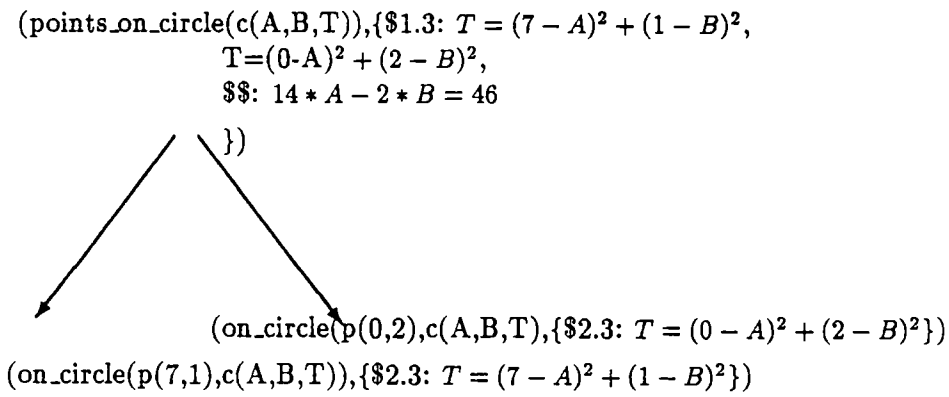
(on_circle(p(2,0),c(A,B,T)),{\$2.3 : $T = (2 - A)^2 + (0 - B)^2$}).

Consider the following extension to the previous example:

points_on_circle(C) :- on_circle(p(7,1),C), on_circle(p(0,2),C).

The query ?- points_on_circle(C)) results in a reduction tree as illustrated below. Here $14 * A - 2 * B = 46$ is the solution to the constraints

$$T = (7 - A)^2 + (1 - B)^2 \text{ and } T = (0 - A)^2 + (2 - B)^2.$$



4 Explanation

In this section we illustrate how CLP reduction trees can be used to generate explanations for constraint computations. Examples of CLP($\mathcal{R}$) computations are used as illustrations.

We concentrate on *how* explanations of successful computations [7]. Explanation can be provided incrementally and the user has the ability to choose the depth of explanation in the explanation system. At any depth the following things are identified and explained:

- Explicit constraints: Explicit constraints could be of the types which were explained earlier.
- Implicit constraints: Implicit constraints occur in the constraint component if any subterm of the head of a clause used in the computation has as principal functor an interpreted function symbol.
- Facts

- Explainable goals : The explanation system is capable of explaining the subgoals. The user can ask for further explanation by typing the query how(*number*) where number is an index to the explainable goal.

We begin with the a constraint explanation for $fib(3,3)$, using the reduction tree in Example 1.

How Explanation:

>>how(fib(3,3))?
fib(3,3) is true with implicit constraint $2: 3=2+1 because
(1) 3>1,
(2) fib(2,2) explainable with constraint $2: 2= 3-1
(3) fib(1,1) is a fact

The user can now instruct the explanation system to further expatiate any of the explainable goals, as follows:

>>how(2)
fib(2,2) is true with implicit constraint $2: 2=1+1
(1) 2>1
(2) fib(1,1) is a fact
(3) fib(0,1) is a fact

Here is an explanation for $foo(10,6)$ from the reduction tree in Example 2.

>> how(foo(10,6))?
foo(10,6) is true with implicit constraint 6 = N1 + N2 because
(1) 10 = 10
(2) N1 > 5
(3) N2 < 10

and 6 = N1 + N2, N1 > 5, N2 < 10 is a solvable system of constraints.

It seems plausible to modify how explanations, giving *whynot* explanations for failed goals. A *whynot* explanation would show the set of unsatisfiable constraints which led to the failure of the query. Consider a slightly modified *foo* predicate - *foo*1, which has a further restriction on $N2$. *foo*1 is defined as follows:

foo1(X,N1+N2) :- X=10, N1> 5, N2 > 0, N2 < 10.

The query $foo1(10,4)$ does not succeed and the following explanation could be provided :

>> whynot(foo1(10,4))?

Explanation:

foo1(10,4) is not true because

(1) 10 = 10,

(2) N1 > 5

(3) N2 > 0

(4) N2 < 10

and 4 = N1 + N2, N1 > 5, N2 > 0, N2 < 10 is not a solvable system of constraints.

The next explanation shows how information about partial solution of constraints can be explained.

>>how(points_on_circle(C))?

Explanation:

points_on_circle(c(A,B,T)) with $14 * A - 2 * B = 46$ is true because

(1) on_circle(p(7,1),c(A,B,T)) fact with $T = (7 - A)^2 + (1 - B)^2$

(2) on_circle(p(0,2),c(A,B,T)) fact with $T = (0 - A)^2 + (2 - B)^2$

and $14 * A - 2 * B = 46$ summarizes

$T = (7 - A)^2 + (1 - B)^2$ *and* $T = (0 - A)^2 + (2 - B)^2$

The explanation provided depends on the underlying program and on the representation of constraints. In the previous explanation the constraint component of the corresponding reduction tree contains implicit components for T in the goal head but the explanation system avoids duplicity of information and does not state the implicit constraints for T again. These examples illustrate explaining constraint computations by traversing the reduction tree, extracting relevant information and presenting it to the user in a meaningful way. More examples and explanations are available in [8].

5 Conclusion

Reduction trees are a generalization of the *proof trees* used to explain conventional Prolog programs. Reduction trees differentiate between unification and constraint solution involving interpreted functions over a domain. Explanations traverse the reduction tree structure to explain CLP programs. Since meta programming facilities available in the current implementation of CLP($\mathcal{R}$) are not adequate to implement a program to construct the reduction tree, the reduc-

tion trees were crafted manually. We feel that reduction trees are well suited for explaining a constraint computation system.

References

[1] Jacques Cohen, "Constraint Logic Programming Languages", Communications of the ACM, July 1990, pages 52-68.

[2] Joxan Jaffar and Spiro Michaylov, "Methodology and Implementation of a Constraint Logic Programming System", Proceedings of the Fourth International Conference in Logic Programming, Melbourne 1987, MIT Press, pages 196-218.

[3] Joxan Jaffar and Jean-Louis Lassez, "Constraint Logic Programming", Proceedings of the 14th ACM Symposium on the Principles of Programming Languages, 1987, pages 111-119.

[4] Nevin Heintze and Spiro Michayov, Peter Stuckey and Roland Yap, "On Meta-Programming in CLP($\mathcal{R}$)", Proceedings of the North American Conference on Logic Programming, Cleveland 1989, pages 52-67.

[5] Pierre Lim and Peter Stuckey, "Meta Programming as Constraint Programming", Proceedings of the North American Conference on Logic Programming, Austin 1990, pages 406-421.

[6] Leon Sterling, "A Meta Level Architecture for Expert Systems". In P. Maes and D. Nardi, editors, *Meta-level Architectures and Reflection*, North Holland 1988, pages 301-313.

[7] Leon Sterling and Ehud Shapiro, "The Art of Prolog", MIT Press 1986.

[8] Leon Sterling and Venkatesh Srinivasan, "Explaining Constraint Computations" - Technical Report TR91-103, Center for Automation and Intelligent Systems Research, Case Western Reserve University, Cleveland, OH 44106.

Layer Constraints in Hierarchical Finite Domains

Philippe Codognet *, **Pierre Savéant** †,
Enrico Maïm †, **Olivier Briche** †

Abstract

We present a framework for handling various kinds of constraints in type hierarchies (taxonomies). In addition to the basic inheritance and equality constraints proposed by the LOGIN language, and disequality, we introduce inheritance, equality and disequality relative to a specific layer in the hierarchy. This allows to both enhance the expressiveness of the language and to design new propagation techniques that we call layer propagation. The underlying idea is to apply the propagation mechanisms of the CHIP language to hierarchical finite domains instead of flat ones.

1 Introduction

Object inheritance has proved to be a useful notion in various programming paradigms. Type hierarchies, or taxonomies, described by an *is_a* relation allow to encode several kinds of information about objects. In Logic Programming, the LOGIN approach [1] consists in integrating inheritance directly into the unification process rather than indirectly in the inference engine (Prolog). This can be seen as a special Constraint Logic Programming (CLP) language [4] dealing with hierarchical constraints over finite lattices. Inheritance (*is_a*) constraints are efficiently handled, thanks to a boolean encoding of the taxonomy, by a kind of *compilation* of the inheritance relation.

The CRL language [5] extends the LOGIN approach, in particular in considering the type constraints in the spirit of the finite-domain constraints of CHIP [6]. CHIP provides efficient constraint handling techniques for *flat* finite-domains, i.e. flat taxonomies representing sets of possible values. This is achieved by having a special low-level encoding of finite-domains and *constraint propagation* techniques. The latter enables constraints to be *active* and to prune the search space in an *a priori* way, eg. forward-checking or looking-ahead techniques.

*INRIA, B.P. 105, 78153 Le Chesnay, FRANCE (codognet@minos.inria.fr)

†SYSECA, 315 bureaux de la Colline, 92213 St Cloud, FRANCE

The idea is to apply CHIP-like technology to hierarchical finite domains (taxonomies). In this way one can benefit both from the ability to reason at several layers of abstraction given by the type hierarchy and from the efficient constraint solving techniques over finite domains. Having a taxonomic structure allows to work at the most abstract level and thus to factorize computations that would have been necessary if we only had a flat structure. We will treat three types of constraints: inheritance constraints, noted X:Y, i.e. the transitive closure of the basic *is_a* relation, equality (common subtype coercion), noted X=Y, and its opposite (no common subtype), noted X≠Y. The first two are already present and efficiently handled in LOGIN, the last will be treated in a way similar to the disequation constraint (≠) of CHIP, by active domain reduction. Moreover, the expressive power of taxonomies and the possibility to reason at different layers of abstraction will amount to the definition of a new type of constraints. We propose an extension of the LOGIN framework which focuses on the notion of *layer*, that can be used both to enhance the expressiveness of the language and to design new propagation techniques that we call layer propagation.
This control mechanism allows the user to tune the granularity at which computation will take place and to save computation work. This control mechanism can indeed be seen as an "implementation" of the principle of reasoning at the higher layer of abstraction in taxonomies.
The new constraints are equality and disequality and membership with respect to a specific layer. Assume for instance that we have a time taxonomy with different granularities considering months, weeks and days. One may for instance want to state that two events happen the same month (but not necessarily the same week or day) or that two events do certainly not happen the same week (even if we don't know which week). This is not feasible with the basic constraints that we have previously presented, but the layer constraints that will be detailed allow this kind of knowledge to be expressed.
The layer propagation technique consists in considering as active only constraints concerning a certain layer (or layers above), while delaying constraints concerning lower layers until necessary.

This paper is organized as follows : the next section presents taxonomies and lattice domains, together with the basic low-level encoding techniques, while section 3 details the different constraints and their operational behavior. A short conclusion ends the paper.

2 Lattice domains

We will present in this section taxonomies and lattice domains and recall the efficient encoding proposed by [2].

2.1 Taxonomies, posets and semilattices

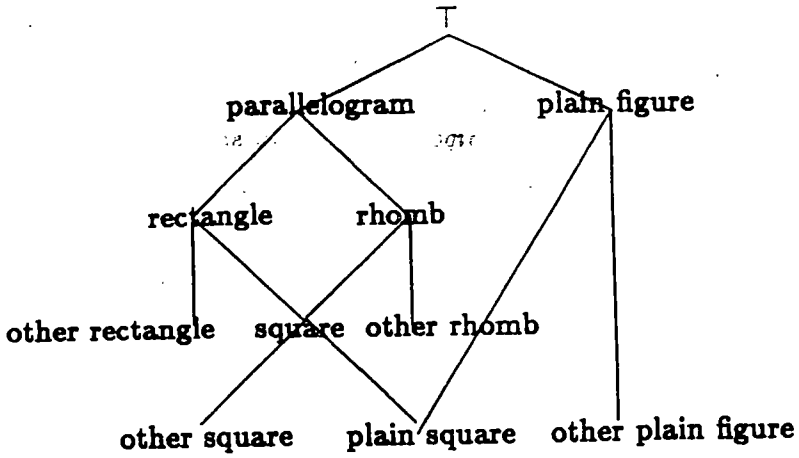
A taxonomy is indeed a *partially ordered set (poset)* whose order relation ($\leq$) is *is_a*, but it is not necessarily a lattice or even a semilattice. The taxonomy may be completed with a top ("universe") and a bottom ("empty") element to ensure that the *greatest lower bound (GLB)* and *least upper bound (LUB)* always exist between any two elements, but nothing ensures that they are unique. As we are interested in inheritance properties, we will only try to give the taxonomy a lower semilattice (LSL) structure. This property can be checked when the taxonomy is given, but this requires from the programmer to specify many pairwise GLBs, that are indeed implicit.

To avoid this task, the taxonomy will be embedded into a LSL structure that is compatible with the original $\leq$ relation, and also contains the necessary GLBs. We will use for this purpose the technique developed for LOGIN and LIFE in [2], which will be detailed in the next section.

The interest in embedding the taxonomy in a LSL structure is that in the latter structure the computation of the GLB, and $\leq$ (is_a) relation, i.e. the basic operation needed for our inheritance properties and constraints, can be made very efficient due to a specific boolean encoding. Indeed these operations are very frequent during a program execution, and it is crucial to have them fast.

2.2 Efficient encoding of semilattices

The semilattice encoding consists in considering the *restricted powerset* RP(L) of a poset L, that is the set of non-empty sets of pairwise incomparable elements of L. Observe that L can be trivially injected in its restricted powerset by identifying a with $\{a\}$. The extra-elements in RP(L) correspond in fact to the "missing" GLBs of L that are needed to form a LSL structure. The elements of the restricted powerset can be encoded as bit-vectors on which the basic operations of computing the GLB and order checking can be efficiently performed. The encoding consists in providing each element with a unique code, in the form of a bit-vector whose size is the number of elements of the initial taxonomy. The extra-codes produced by the encoding correspond to the specific elements of the restricted powerset . They are precisely the extra LUBs required for the embedding of the taxonomy (poset) in a LSL. Consider the poset:



A first method is given by taking the matrix of the reflexive transitive closure of the *is_a* relation, where 1 in row i and column j indicates that j inherits from i. For each element of the poset, the corresponding row of the matrix can be seen as a bit vector code of this element. There is another way of realizing a similar encoding: instead of computing the transitive closure matrix, one can simply make a poset traversal layer by layer, starting from the bottom element, and assigning codes to the elements. The method is rather natural. Each node of the poset has a power of 2 that identifies it. For the reflexive transitive approach, the node is then represented by the union of its power of 2 and the powers of 2 of all nodes that are less than it (the partial order being defined by the *is_a* relation). To get codes more compact, we get rid of useless powers of 2 which is the case every time the GLB is unique. Algorithms are described in [2]. On the previous example, the encoding is as follows.

Node	Code
bottom	00000
other plain figure	00001
plain square	00010
other square	00100
other rhomb	10000
square	00110
other rectangle	01000
rhomb	10110
rectangle	01110
plain figure	00011
parallelogram	11110
universe	11111

The main interest of this encoding is to allow efficient computation of the basic operations that are needed to solve the type constraints. Indeed GLBs can be computed by simply taking the bitwise AND of the corresponding elements. Consider

for instance *plain square*, the GLB of *rectangle* and *plain figure*, its code is the AND of the code of *rectangle* and *plain figure*.

The semilattice embedding allows the mapping of union, intersection and complement of types respectively into binary *or*, *and* and *complement* of their corresponding codes. The complemented object will be used when dealing with the disequation constraints, for representing objects with a negative part. Such an object can be represented by t_1 - t_2, that is interpreted as $t_1 \cap \overline{t_2}$ and is encoded as $\gamma(t_1) \wedge \overline{\gamma(t_2)}$, γ being the encoding function.

3 Constraints

3.1 basic constraints

We will now detail the handling of our type constraints, that is X:Y (X inherits from Y), X=Y (type equality) and disequality : X≠Y (instance disequality).

3.1.1 X = Y

Operationally, this constraint is treated by creating a new variable Z with a type domain which is the GLB of the type of X and Y, and by binding both X and Y to Z. Observe that this constraint is equivalent to the conjunction $X : Y, Y : X$.

This operation can be efficiently implemented by assigning to the bitvector encoding of the domains of Y and X the bitwise AND on these codes, corresponding to their GLB.

3.1.2 X : Y

Operationally, it is treated as follows. If X is a (direct or indirect) subtype of Y, this constraint succeeds. Otherwise, a new variable Z is created whose domain is the GLB of X and Y. If the domain of Z is reduced to $\perp$, then the constraint fails, and the whole system is unsatisfiable. If this domain is non-empty, then X is bound to Z to ensure that X is included in the domain of Y.

Thanks again to the encoding, this operation corresponds to a simple logical operation on bit-vectors : one simply assign to the code of X the AND of those of X and Y and checks that it is not zero (code of $\perp$).

3.1.3 X≠Y

As we want to treat this constraint by forward checking techniques, the meaning of X≠Y will be that X and Y denote two different *basic types*. A basic type is a type covering the bottom root, i.e. that has no other subtype than bottom. Basically, forward checking consists in delaying constraints until one of their variables is bound to a value. A disequation is treated by removing this value from the domain of the other variable. The search space is thus pruned in an *a priori* way. This explains the restriction to basic types, which are singletons whereas other types denote sets of types.

Operationally removing the value of X from the domain of Y consists in updating the domain of Y by taking the bitwise AND with the complement of the domain of X: $\gamma(domain(Y)) \leftarrow \gamma(domain(Y)) \wedge \overline{\gamma(domain(X))}$, γ being the encoding function.

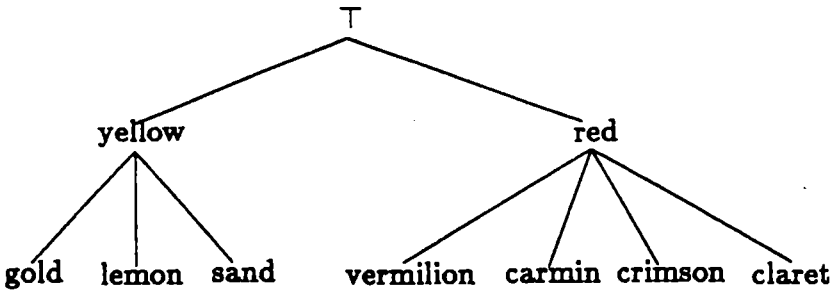
3.2 Layer Constraints

Hierarchical domains allow to define layers of abstraction, and we will see the use of this concept to extend both the expressiveness of our constraint language (layer constraints) and the constraint solving process (layered propagation).
The new constraints are equality and disequality and membership with respect to a specific layer. Assume for instance that we have a time taxonomy with different granularities considering months, weeks and days. One may for instance want to state that two events happen the same month (but not necessarily the same week or day) or that two events do certainly not happen the same week (even if we don't know which week). This is not feasible with the basic constraints that we have previously presented, but the layer constraints that will be detailed below allow this kind of knowledge to be expressed.
The basic idea of the layered propagation technique is to control constraint propagation by taking into account a layer upon which the triggering of a constraint propagation depends. When constraint propagation is done at layer L, only constraints concerning layer L or above are woken up and used in propagation, other constraints are delayed until their layer is considered. The programmer can hence "tune" the constraint solving process by stating the triggering layer, and refine repeatidly the answer by going to a lower layer if necessary. This mechanism that considers only constraints and domains above a certain layer is also important for efficiency, as staying at the higher layers allows a bigger pruning in the domains, and as the computation work related to (too) specific constraints is delayed until it is really needed and can be saved if not necessary.

3.2.1 Layers

A *layer* is a set L of pairwise incomparable types such that all elements of L are at the same rank. The rank of a node is the length of the longest path from the top root to this node.

For the sake of simplicity we allow to give a layer a symbolic name, as shown by the following example:
Consider a simple classification for a chart of colors.



Possible layers on this example are {yellow,red} that we call the *color* layer and {gold,lemon,sand,vermilion,carmin,crimson,claret} that we call the *shade* layer.
We will say that a type t is *below* (resp. *above*) a layer L if $\exists l \in L/t : l \wedge t \neq l$ (resp. $\exists l \in L/l : t \wedge t \neq l$). We say that t is *at* layer L if $t \in L$.

3.2.2 X $=_L$ Y

This constraint states that, whatever the domains of X and Y are, they have an upper bound (common ancestor) which is at or below layer L. Considering the taxonomy of the color example, X $=_{color}$ Y means that X and Y have the same color (i.e. red, yellow, ...), but not necessarily the same shade. For instance, we have gold $=_{color}$ lemon.

This constraint is handled as follows :
let t = GLB(X,Y). If t is at or below layer L, then the constraint can be treated : X is bound to a new variable X1 whose domain is t. Note that X and Y are not bound to each other as for the basic = constraint.
If t is above L, the the constraint is delayed until this condition is satisfied. This constraint will hence be woken up each time the domain of X or Y will be changed in order to check the above condition.

3.2.3 X$\neq_t$Y and layered forward checking

Back to the color example, suppose that one wants to express that X and Y are different colors (one red, the other yellow), and not simply of different shades. This was suggested by a map coloring problem. Indeed what we want to express is that X and Y are different at the color layer. Therefore, the disequation operator has to be indexed by the layer. On the example, the *color* layer is the set {*yellow, red*} and our constraint will be written X$\neq_{color}$Y.

This constraint will be handled by *layered forward checking*, i.e. forward checking triggered as soon as one variable is bound to a type at or below the indicated layer.

Suppose on the color example that X is bound to *red* and Y to *color*, then the constraint is woken up, that is *red* is removed from the domain of Y which is now reduced to *yellow*. Variables can further be bound to more specific shades, the constraint will always be enforced.

At the implementation level, it works exactly the same as the simple case : $\gamma(domain(Y)) \leftarrow \gamma(domain(Y)) \wedge \overline{\gamma(domain(X))}$.

3.2.4 $X:_L Y$

This constraint ensures that X is a subtype of Y whose domain is at layer L or below. Taking the same example again, $X:_{shade} Y$ constrains X to be a subtype of Y which is a shade (vermilion, sand, ...) and not just a color (red, yellow, ...). Observe that the constraint $X:_\top Y$ ensures that the domain of X is at or below layer L.

This constraint is handled as X:Y with an extra checking : if the domain of X is below L, then the constraint is satisfied, else the constraint is delayed and will be woken up each time the domain of X will be changed in order to check this condition.

4 Conclusion

We have presented hierarchical finite domains, i.e. taxonomies, and their associated constraints. Basic constraints such as equality, membership or disequality are well known, thanks to the LOGIN language. We proposed an extension which focuses on the notion of layer, that can be used both to enhance the expressiveness (considering equality or disequality with respect to a given layer) and for designing new propagation techniques based on the triggering of constraints with respect to a given layer. This control mechanism allows to stay at the higher level of abstraction and to save computation work.
We have based our approach on the lattice embedding of taxonomies proposed by LOGIN, as it provides an efficient low level encoding of domains and efficient basic constraint handling, and it would be interesting to consider other algebraic structures. For example, for applications dealing with temporal reasoning at different granularities [3], one can consider the algebra of time intervals, and an interesting topic is to see how our layer constraints can be adapted to fit into this framework.

This work was done in the framework of the development of the CRL language in the European Esprit II (♯ 2409) project.

References

[1] Aït-kaci Hassan and Nasr Roger, "LOGIN: A logic programming language with built-in inheritance", *The Journal of Logic Programming*, **3** (3), 1986.

[2] H. Aït-Kaci, R. Boyer, P. Lincoln and R. Nasr : "Efficient implementation of lattice operations", Transactions on Programming Languages and Systems, vol. 11 n° 1, January 1989.

[3] Evans Chris, "The Macro-Event Calculus: Representing Temporal Granularity", *proceedings of the 1st Pacific Rim International Conference on Artificial Intelligence*, Nagoya, Japan, 1990.

[4] J-L. Lassez and J. Jaffar : "Constraint Logic Programming", proceedings of POPL 87, Munich 1987.

[5] Maïm Enrico, "CRL: Common Representation Language", *Proceedings of the Second International Conference on Software Engineering and Knowledge Engineering*, Skokie, Illinois, 1990.

[6] P. Van Hentenryck : "Constraint Satisfaction in Logic Programming", MIT Press, 1989.

Restriction Site Mapping as a Constraint Satisfaction Problem

ROLAND YAP
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights
NY 10598
U.S.A.

January 29, 1991

Abstract

The Restriction Site Mapping problem is an important computational problem in molecular biology. In this paper, we show how to formulate this problem as a dynamic constraint satisfaction problem. This leads to an elegant solution for solving linear restriction site mapping problems for two enzymes using the constraint logic programming language CLP($\mathcal{R}$). The solution here is different from classical CSP techniques as the system of constraints here is dynamic and instead of finding a satisfying assignment of values for the variables, we obtain a satisfying set of constraints.

1. Introduction

Restriction site mapping is an important problem in molecular biology. An introduction to the problem is contained in [3]. In this section, we will briefly review some of the molecular biology background of the problem. Knowledge of these details is not necessary for understanding the problem which is formulated abstractly in section 2.

Restriction site mapping is an important tool used in sequencing and cloning DNA. A DNA molecule can be cut into fragments using an enzyme called a *restriction enzyme*. The restriction enzyme recognises specific patterns in the DNA sequence and cuts the molecule at those particular places, which are known as *restriction sites*. The *restriction site mapping problem* then is to reassemble together the original molecule given those fragments. The reconstructed molecule or *restriction map* serves as a gross description of the original molecule since the position and patterns of the restriction sites are known.

The DNA molecules which are to be mapped come in two kinds, linear molecules which can be thought of as being a string over a four letter alphabet or a circular molecule. Restriction enzymes can be applied to the molecule in a number of ways. Application of a single enzyme alone is known as a *single digest* and applying two enzymes together is a *double digest*. A double digest cuts the DNA at both the restriction sites of the two enzymes. Figure 1, shows the cut sites on a linear molecule with enzymes A and B. The thick horizontal lines are the fragments

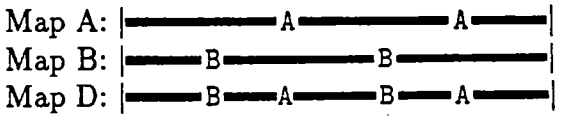


Figure 1: A linear restriction map

which would be produced by applying the enzymes A, B and the combination of A and B (the double digest D). The sites are the letters A and B separating the thick lines. Reconstruction of the maps for A, B and the double digest D given the fragments would produce the figure above. Basically the enzymes cut up the DNA at the sites A in map A and B in map B. The map of the double digest is obtained when the A and B maps are superimposed.

The double digest constrains the way fragments can fit in the individual single digest maps and together these determine the total map. The fragments themselves are obtained through experiments as fragment lengths. Because the lengths are experimental data, the mapping problem is also complicated by the presence of

errors in the fragment lengths (usually percentage error). In general, there may be a number of consistent maps for the same data. The presence of errors means that the fragments may not match up exactly, for example the sites in Figure 1 match up by aligning vertically since there are no errors. It also tends to increase the number of solutions and thus it is important to treat fragment errors uniformly in order to avoid missing possible solutions.

This paper describes a solution to the restriction site mapping problem for two enzymes in the presence of errors for linear molecules. The solution is formulated in terms of constraints which are built up incrementally using the notion of consistent map prefixes. This leads to a straightforward implementation in the constraint logic programming language CLP($\mathcal{R}$)[10].

2 Formulation of the problem constraints

In this section we develop the conditions for a consistent set of restriction maps for a linear map with two enzymes, call these two enzymes A and B. Throughout this paper we will use the symbol $\mathcal{A}$ to denote things which are related to the single digest with enzyme A, $\mathcal{B}$ for single digests of B and $\mathcal{D}$ for the double digest. Recall that we two enzymes, there is either the single application of the enzyme to produce digests $\mathcal{A}$ and $\mathcal{B}$ or the combined application of A and B to produce the digest $\mathcal{D}$.

Firstly, let $\mathcal{A}$= $\{a_1, a_2, \ldots, a_{n_A}\}$ and $\mathcal{B}$= $\{b_1, b_2, \ldots, b_{n_B}\}$ be the set of true single digest fragment lengths, and $\mathcal{D}$= $\{d_1, d_2, \ldots, d_{n_D}\}$ be the set of true double digest fragments, where n_A, n_B and n_D are their number respectively. Note that we distinguish the true possibly unknown lengths from the observable lengths. A consistent map of the whole molecule is some permutation of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{D}$ which satisfies the conditions below. We will say map $\mathcal{A}$ to mean the particular map involving only the restriction sites of $\mathcal{A}$. A map like map $\mathcal{A}$ is simply a string consisting of all the fragments of $\mathcal{A}$ permuted in some fashion. Also, define the *prefix length* of $\mathcal{A}$, $\mathcal{A}_j$, to be the length of a prefix of the map of $\mathcal{A}$ with j fragments, and similarly for $\mathcal{B}_j$ and $\mathcal{D}_j$. The conditions for map consistency are:

- The total length of all digests are equal/consistent:

$$\sum_{i=1}^{n_A} a_i = \sum_{j=1}^{n_B} b_j = \sum_{k=1}^{n_D} d_k = L \tag{1}$$

where L is the length of the original DNA molecule.

- The single digests are composed of double digest fragments, i.e. the length of every prefix of $\mathcal{A}$ is compatible with a prefix of $\mathcal{D}$:

$$\forall i \exists j : \mathcal{A}_i = \mathcal{D}_j \text{ and } \forall i \exists j : \mathcal{B}_i = \mathcal{D}_j \quad (2)$$

  Also, we will write $\mathcal{A}(k) \equiv \forall i : 1 \leq i \leq k,\ \exists j\ A_i = D_j$ (the first half of condition (2)), and similarly $\mathcal{B}(k)$ for the second half.

- Every end of a double digest fragment must originate in either map $\mathcal{A}$ or map $\mathcal{B}$ (except for the the start and end piece which has only 1 cut site):

$$\forall i \exists j, k : \mathcal{D}_i = \mathcal{A}_j \text{ or } \mathcal{D}_i = \mathcal{B}_k \quad (3)$$

- Since there are errors in the fragments, each a_i the unknown true length has an associated upper bound (a_i^u) and a lower bound (a_i^l) for the length. For every fragment, this gives:

$$\forall i : a_i^l \leq a_i \leq a_i^u \text{ and } \forall i : b_i^l \leq b_i \leq b_i^u \text{ and } \forall i : d_i^l \leq d_i \leq d_i^u \quad (4)$$

In summary, the consistency conditions above enforce alignment for the restriction sites so that a site on the double digest lies on a single digest and vice versa. Because of the presence of errors, checking a valid permutation of $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{D}$, requires checking all the consistency conditions above and simply lining up the fragments as in the exact example of figure 1 is not sufficient.

3 Solving the problem constraints

Before we look at constraint approaches to the problem, we will briefly mention some of the other approaches in the literature. Many of the methods like [2, 6, 7, 15, 14] are based on the combinatorial combination of compatible partitions but usually only local consistency checks are applied. Other statistical methods such as least squares [13] and simulated annealing [8] have also been used. However while many of these approaches are similar in spirit they handle errors in different and somewhat ad-hoc ways. Also some of the methods like the statistical ones have the disadvantage of not being complete and only produce some of the solutions. The approach in [1] is more similar to the work here except that they focus at a low level on consistent loops and loop checking. The approach in this paper is to view the problem in terms of constraints and utilise constraint solving techniques. We suggest that this is a natural way of solving and analysing restriction site mapping because reasoning at the level of the constraints is much simpler.

Restriction site mapping is a computationally intensive problem and can be shown to be NP-complete [8]. Thus a search guided by CSP techniques for dealing with the constraints is a natural approach. There are a number of different possibilities to using the conditions of section 2. The classical view of constraint satisfaction deals with pre-specified constraints over a known domain, that is the constraints are static and global. Conditions (2) and (3) define a consistent map in terms of a valid permutation of the fragments, but this permutation is not known and as such is harder to state statically. The other conditions, being global can of course be stated statically. One way of dealing with the unknown ordering is to factor it in by using a permutation matrix. So the prefix length $\mathcal{A}_i$ can be written as $\mathcal{A}_i = \sum_{j=1}^{i} P_j \bar{A}$ where $\bar{A}$ is a vector of the fragments and P_j is the j-th row of a permutation matrix. A permutation matrix is simply an identity matrix where the rows and columns have been permuted, e.g. $P_{ij} = (0 \vee 1) \wedge \sum_i P_{ij} = 1 \wedge \sum_j P_{ij} = 1$. With this formulation, conditions (2) and (3) can be written statically using prefix lengths and the permutation matrix.

Such a static formulation has a number of disadvantages. Firstly the number of variables is now increased by $\mathcal{O}(n^2)$. The problem is now in the form of a non-linear programming program (the true fragment lengths a_i are not known) and nothing is gained from this transformation as it makes little use of the constraints to prune the solution space. This is basically a static transformation of a dynamic problem.

Another other major difference lies in the domain of the variables in the constraints. Typically CSP formulations deal with finite domains for the variables. The variables here are the fragment lengths which are in principle integral. However the cardinality of the domain of a single variable can range from the hundreds to the thousands. This suggests that simply using the variables as ranging over a finite set of values may lead to an explosion in the size of the search space. Also we are interested in a description of the solution and not necessarily in the values for the variables since there would be a large number of solutions with different values which can be more concisely described by a set of answer constraints. Thus, we choose here to use the infinite domain of the real numbers instead for the constraints. Also changing to the reals relaxes the problem constraints. Since the problem constraints involve arithmetic, such constraints with the real numbers are generally easier to solve than in the integers. It is also more convenient to use the reals since the lengths are from experimental measurements.

3.1 Using dynamic constraints

A more problem orientated approach is to consider it as dynamic constraint problem where the conditions of section 2 are gradually applied to build up the map incrementally. Once a prefix segment has been determined the constraints on that segment are also determined since that part of the permutation is known. The constraints are dynamic as they are added incrementally by extending the segment prefixes. Constraint logic programming (CLP[9]) languages such as CLP($\mathcal{R}$), Prolog III [4] and CHIP [5] are naturally suited for such a dynamic constraint satisfaction problem.. In particular, CLP($\mathcal{R}$) is very appropriate as it deals with arithmetic constraints on the real numbers. In CLP, constraint solving/satisfaction is dynamic and constraints are added incrementally during forward execution and deleted appropriately by backtracking. The difference with the classical CSP approach above is that the constraints are no longer pre-specified and satisfying a set of constraints corresponding to a prefix will lead to a longer prefix with more constraints being added and so on. The other advantage of solving such combinatorial constraint problems in a CLP language arises from the fact that it is a programming language and is thus more flexible for encoding the problem. The program can dynamically generate and select which constraints to solve, problem heuristics can be incorporated easily and being a declarative logic programming system it is flexible and can be used in a number of different ways, e.g. fragments can be specified or not, partial answers can be used, etc.

We can now formulate the constraints dynamically in the following way. The maps are built up incrementally by adding one fragment at the time and at the same time applying the consistency contraints resulting from those fragments. Consider $\mathcal{A}_1$, $\mathcal{B}_1$ and $\mathcal{D}_1$, the first segments of the map. We have from conditions (2) and (3) the initial starting condition:

$$\begin{array}{lcl} \mathcal{A}_1 = \mathcal{D}_1 & \wedge & \mathcal{B}_1 \geq \mathcal{D}_1 \\ & \text{or} & \\ \mathcal{B}_1 = \mathcal{D}_1 & \wedge & \mathcal{A}_1 \geq \mathcal{D}_1 \end{array} \tag{5}$$

Condition (3) is satisfied but condition (2) may not be satisfied with the current prefixes obtained so far, e.g. $\mathcal{A}_1 > \mathcal{D}_1$ but $\mathcal{A}_i = \mathcal{D}_j$ is required. That part of (2), say $\mathcal{A}(1)$ may be satisfied by adding more $\mathcal{B}$ and $\mathcal{D}$ fragments while maintaining all other conditions. Thus by adding fragments simultaneously along the double digest and one of the single digests we can maintain the following invariant:

$$\left(\begin{array}{lcl} \mathcal{A}_i = \mathcal{D}_j & \wedge & \mathcal{B}_k \geq \mathcal{D}_j \\ & \text{and} & \\ \mathcal{A}(i) & \wedge & \mathcal{B}(k-1) \end{array}\right) \bigvee \left(\begin{array}{lcl} \mathcal{B}_k = \mathcal{D}_j & \wedge & \mathcal{A}_i \geq \mathcal{D}_j \\ & \text{and} & \\ \mathcal{B}(k) & \wedge & \mathcal{A}(i-1) \end{array}\right) \tag{6}$$

At every step this may lag behind satisfying condition (2) by one single digest

fragment. Finally when all the fragments have been placed we have the complete and consistent map.

It is easy to write the corresponding CLP($\mathcal{R}$) program to generate the map in this fashion. The domain variables are simply the fragment lengths a_i, b_i and d_i which are real numbers constrained to lie within the intervals of condition (4). Fragments are first chosen corresponding to the initial starting condition (5). Then the map is built up incrementally by extending the map prefixes one at a time along the double digest D and one of the single digests, A or B, until the complete map has been constructed. The map is extended by choosing a d_i and one of a_j or b_k which maintains the invariant. The constraints resulting from the extension are added to the system of constraints of the earlier map. These are simply the appropriate part of the disjunction of invariant (6). Note that we choose among fragments and not fragment values. The program obtained is quite straightforward and is only about a page in length omitting auxiliary routines. We also add reflection symmetry constraints and heuristics on variable ordering.

4 Discussion

In this paper, we have sketched a dynamic constraint formulation for solving linear restriction maps. This can be easily extended to circular maps by transforming a circular map to a linear one and imposing additional constraints.[1] Comparing the solution here with other approaches in the literature, this is much easier to work with and understand because it is formulated in terms of the problem constraints. A programming language like CLP($\mathcal{R}$) allows such constraints to be stated directly and be easily combined in a dynamic fashion. This problem is an example of a CSP-like problem which can be described easily in terms of how constraints interact with each other in a dynamic way whereas viewing it as a global set of static constraints to be solved is less useful and does not take advantage of knowledge of the problem. In contrast to CSP-techniques for manipulating values of variables to maintain consistency such as arc and path consistency [11] or waltz filtering [16], the approach here manipulates the problem constraints themselves. So the approach here is to select valid constraints to describe the map rather than selecting valid values for variables. Also in the final solution, if the fragment lengths are only constrained to lie within an error interval then the problem variables do not have definite values even though a solution has been obtained i.e. the answers are not ground. Instead the answer is a simpler set of constraints which describe the solution space, i.e. possibly new tighter fragment intervals and the summation of

[1] [17] describes solving linear and circular maps in CLP($\mathcal{R}$).

	Map 1	Map 2
No. of A	3	4
No. of B	4	6
No. of D	6	9
No. of solutions	16	12
Fragments placed	285	2491
Time (RS/6000)	2.6s	48.5s

Table 1: Run-time statistics on two sets of data

	Map 1	Map 2
Fragments placed	721	2491
Time (RS/6000)	3.3s	31.8s

Table 2: Run-time statistics for the relaxed problem

some fragment variables on one digest equated to those on another digest.

Preliminary experiments with random and experimental data indicate that the problem constraints do prune the search space considerably. Table 1 shows some statistics on real experimental data for two linear maps which has been obtained using an experimental version of CLP($\mathcal{R}$). The fragments placed column refers to the number of fragments which were successfully placed when trying to extend the map, i.e. the fragments which were consistent prefixes at that point. This is a measure of the size of the actual search space of the problem considered. The worst case size of the search space is $\mathcal{O}(n_A!n_B!n_D!)$ while the actual number of fragments placed is relatively small which indicates that the problem constraints are fairly strong but the amount of time for constraint solving is also significant. One heuristic for reducing the amount of work required in constraint solving is to relax the program constraints. This leads to faster solving at the expense of a potentially greater search space since the problem conditions have been weakened. For example, the program can be relaxed by dropping condition 1 which requires the total lengths to be compatible. Figure 2 gives the corresponding relaxed version of the problems from Figure 1. We can see that with map 1, dropping the length requirement doesn't help as it increases the amount of time required while with map 2, the time required is much less. The reason for this is that in map 1, condition 1 was a strong constraint because the total lengths of the different digests was quite different and thus constrained the individual fragments somewhat, while with map 2 the total lengths were sufficiently similar. This can be readily seen in the fact that the size of the search space increased with Map 1 while Map 2 was unchanged. So this heuristic can be useful depending on the nature of data being used. Finally we note that this problem is one of the few "real" constraint satisfaction problem which is easily scalable and as such may be a useful problem for investigating

search techniques in the fashion that N-queens is used as a testbed.

References

[1] Allison, L. and Yee, C. N., "Restriction site mapping in separation theory", *CABIOS* (also called *Computer Applications for Biosciences*) 4, 1988, pp 91–101.

[2] Bellon, B., "Construction of restriction maps", *CABIOS* 4, 1988, pp 111–115.

[3] Bishop, M. J. and Rawlings C. J., "Nucleic Acid and Protein Sequence Analysis: A Practical Approach", IRL Press, 1987, Oxford, chapter 6.

[4] Colmerauer, A. "An Introduction to PROLOG-III", *Communications of the ACM* 33(7), 1990, pp 69–90.

[5] Dincbas, M., Van Hentenryck, P., Simonis, H. and Aggoun, A., "The Constraint Logic Programming Language CHIP", *Proceedings of the* 2^{nd} *International Conference on Fifth Generation Computer Systems*, Tokyo, pp 249-264.

[6] Dix, T. I. and Kieronska, D. H., "Errors between sites in restriction site mapping", *CABIOS* 4, 1988, pp 117–123.

[7] Fitch, W. M., Smith, T. F. and Ralph, W. W., "Mapping the order of DNA restriction fragments", *Gene* 22, 1983, pp 19–29.

[8] Goldstein, L. and Waterman, M. S., "Mapping DNA by Stochastic Relaxation", *Advances in Applied Mathematics* 8, 1987, pp 194–207.

[9] Jaffar, J. and Lassez, J-L., "Constraint Logic Programming", Technical Report 86/73, Dept. of Computer Science, Monash University, June 1986. (An abstract appears in the *Proceedings* 14^{th} *ACM Symposium on Principles of Programming Languages*, Munich, January 1987, pp 111–119.)

[10] Jaffar, J., Michaylov, S., Stuckey, P. J. and Yap, R. H. C., "The CLP($\mathcal{R}$) Language and System", IBM Research Report, RC 16292 (#72336), November 1990 (a very early version of this paper appears in *Proceedings* 4^{th} *International Conference on Logic Programming*, MIT Press, June 1987, pp 196–218).

[11] Mackworth, A., "Consistency in networks of relations", *Artificial Intelligence* 8, 1977, pp 99-118.

[12] Nolan, G. P., Maina, C. V. and Szalay, A. A., "Plasmid mapping computer program", *Nucleic Acids Research* 12, 1984, pp 717-729.

[13] Pearson, W. R., "Automatic construction of restriction site maps", *Nucleic Acids Research* 10, 1982, pp 217–227.

[14] Stefik, M., "Inferring DNA Structures from Segmentation Data", *Artificial Intelligence* 11, 1978, pp 85–114.

[15] Tufferey, P., Dessen, P., Mugnier, C. and Hazout, S., "Restriction map construction using a 'complete sentences compatibility' algorithm", *CABIOS* 4, 1988, pp 103–110.

[16] Waltz, D., "Understanding line drawings of scenes with shadows", In *The Psychology of Computer Vision*, P. H. Winston, Ed. McGraw-Hill, 1975, New York.

[17] Yap, R., "Restriction Site Mapping in CLP($\mathcal{R}$)", draft manuscript.