A COMPARISON OF ATMS AND CSP TECHNIQUES

Johan de Kleer
Xerox Palo Alto Research Center
3333 Coyote Hill Road, Palo Alto CA 94304

Abstract

A fundamental problem for most AI problem solvers is how to control search to avoid searching subspaces which previously have been determined to be inconsistent. Two of the standard approaches to this problem are embodied in the constraint satisfaction problem (CSP) techniques which evolved from vision tasks and assumption-based truth maintenance system (ATMS) techniques which evolved from applying constraint propagation techniques to reasoning about the physical world. This paper argues that both approaches embody similar intuitions for avoiding thrashing and shows how CSPs can be mapped to ATMS problems and vice versa. In particular, Mackworth's notions of node, arc, and path consistency, Freuder's notion of it-consistency, and Dechter and Pearl's notion of directed K-consistency have precise analogs in the ATMS framework.

1 Introduction

Both constraint satisfaction techniques and truth maintenance systems are used widely in artificial intelligence applications. A variety of refinements have been developed to avoid the thrashing behavior that often accompanies them. This paper demonstrates that the intuitions underlying these refinements are essentially similar. In particular, Mackworth's [13] notions of node, arc, path consistency, Freuder's [11] inconsistency, and Dechter and Pearl's [1] directed k-consistency for constraint satisfaction problems(CSPs) and techniques for assumption-based truth maintenance (ATMS) [4, 5] are encompassed by a few propositional resolution rules. This paper shows that in many cases the constraint networks produced by the standard local consistency algorithms are identical to ones produced by the ATMS for the same input constraint network. A longer version of this paper [10] explores these issues in far greater detail.

This paper details how both CSP techniques and ATMS's are based on a few resolution rules. By outlining the relationship between these two sets of techniques, a connection in terminology and algorithms is established between two distinct subfields of artificial intelligence. This holds the promise that any advance in one subfield is easily communicated to the other. At the moment both have incorporated essentially the same intuitions. However, we hope that this synthesis will lead to faster progress on these issues of mutual interest.

1.1 Search vs. Inference

Although CSP and ATMS tasks seem, on the surface, to be very different, both require combining two distinct modes of reasoning to minimize overall computational cost to complete tasks. In the following sections we define more precisely what ATMS's and CSPs are. Before diving into those details, we address the frameworks within which CSP and ATMS techniques are used.

CSPs are typically solved by backtrack search. However, as Mackworth [13] demonstrated, backtracking suffers from numerous maladies often making it "grotesquely inefficient." Mackworth pointed out that if the constraints are preprocessed then many of backtracking's maladies can be eliminated. This preprocessing involves tightening (via local consistency algorithms) the initial constraints so that backtracking detects futile states earlier leading to fewer backtracks. Freuder [11] generalized Mackworth's results and showed that with sufficient preprocessing most or all of the futile backtracks can be avoided. However, the expense of the preprocessing to avoid all futile backtracks typically outweighs the cost of the futile backtracks in the first place. A fundamental issue for CSP solvers is to determine how much preprocessing is necessary to minimize total effort (preprocessing plus backtracking).

An ATMS task cannot be so cleanly stated as a CSP. Instead an ATMS is always used interactively with some kind of inference engine. The inference engine is roughly analogous to the backtracker while the ATMS is roughly analogous to the module which enforces CSP local consistency. The inference engine supplies the ATMS with a stream of clauses (as opposed to a CSP where the input problem is fixed) and queries about those clauses. The most typical query the inference engine makes is whether a particular set of propositional symbols is consistent (i.e., is part of a global solution). Ensuring that the ATMS never replies that a set is consistent when it is not is computationally disastrous in practice. However, answering that a set is consistent when in fact it is not results in futile inference engine effort on that set of assumptions.

We thus see that ATMS and CSP techniques embody the same computational trade-offs. Futile backtracking is analogous to futile inference engine work, but avoiding all futile backtracking or all futile inference engine work is too expensive also. Thus, the CSP local consistency algorithms are motivated by exactly the same set of problem solving concerns as the ATMS algorithms. But even more surprising, most of the CSP local consistency algorithms themselves have almost identical analogs in the ATMS framework.

Due to the nature of the ATMS task, all ATMS algorithms are incremental. CSP techniques are typically used to solve completely specified problems, and need not be in-

cremental. However, many of the CSP local consistency algorithms can easily be made incremental. (Although some CSP algorithms take explicit advantage of the fact that the problem is completely specified a priori.) ATMS techniques are more general, not because of their inherent incrementality, but rather that they can directly solve any task stated propositionally. For example, every CSP can be conveniently expressed as an ATMS problem, but not every ATMS problem can be conveniently expressed as a CSP. This gives the ATMS the flexibility that it can intersperse solving a CSP with other kinds of propositional reasoning. The the local consistency intuitions underlying CSP techniques thus have a more general manifestation within the ATMS framework.

2 Definitions

2.1 CSPs

For more details, Mackworth [14] presents an excellent introduction to CSPs. A constraint satisfaction problem (CSP) is specified by a set of variables $x_1, \ldots, x_n$ and a set of constraints on subsets of these variables limiting the values they can take on. $C_{ijk\ldots}$ notates the constraint among variables $x_i, x_j, x_k, \ldots$.

If the constraint places no limitations on its variables, it is called *universal*. By convention, universal constraints are not explicitly represented. Every variable x_i is restricted to a finite set D_i of values $\{a_{i1}, \ldots, a_{in_i}\}$. The language in which these constraints are stated, or the costs of determining whether a set of values satisfies a constraint are typically not addressed in the CSP literature. We address those in section 10.

The constraint satisfaction task consists of finding all sets of values $a_{1j_1}, \ldots, a_{nj_n}$ for $x_1, \ldots, x_n$ that simultaneously satisfy all the given constraints. Two CSPs are *equivalent* if they both have the same set of solutions. Unlike much of the CSP literature we always consider the general case where the constraints can be n-ary and asymmetric.

Consider the following example from Freuder [11]. We are given the variables x_1, x_2, x_3 with domains $D_1 = \{a, b\}$, $D_2 = \{e, f\}$, and $D_3 = \{c, d, g\}$ and binary constraints (specified extensionally by their allowed tuples) $C_{12} = \{be, bf\}$, $C_{13} = \{bc, bd, bg\}$, and $C_{23} = \{cd, fg\}$. This CSP has two solutions:

$$x_1 = b, x_2 = e, x_3 = d,$$

$$x_1 = b, x_2 = f, x_3 = g.$$

Every CSP is characterized by a *constraint network* which, in general, is a hypergraph in which the vertices represent variables and the hyperedges represent constraints. All the CSP local consistency algorithms operate by transforming constraint networks into equivalent but (hopefully) simpler ones.

2.2 ATMS

The input to the ATMS is a set of propositional literals $\mathcal{L}$, a set propositional clauses $\mathcal{C}$ constructed from those literals, and a distinguished set of assumptions $\mathcal{A} \subset \mathcal{L}$. The task of the ATMS can be succinctly characterized as finding all prime implicates mentioning at most one non-assumption literal (i.e., in $\mathcal{L} - \mathcal{A}$) [16]. However, for the purposes of this paper we characterize (equivalently) the ATMS in the terms of [4].

An *environment* is a subset of $\mathcal{A}$. An environment E is inconsistent (called *nogood*) if the union of E with $\mathcal{C}$ is not satisfiable. Nogood X is subsumed by nogood Y if the assumptions of Y are a proper subset of those of X. A *choose* is a clause whose literals are all assumptions and is often written *choose*$\{A_1, \ldots, A_m\}$. Choose R is subsumed by choose S if the assumptions of S are a proper subset of those of R. A literal l is said to *hold* in environment E if l follows from the union of E and $\mathcal{C}$ under the rules of propositional logic. A nogood is *minimal* (with respect to subset) if it contains no others as a subset.

The ATMS problem is to find all minimal nogoods and to identify for every literal $l \in \mathcal{L}$ a set of environments $\{E_1, \ldots, E_k\}$ (called the *label*) having the four properties:

1. [Soundness.] l holds in each E_i.
2. [Consistency.] E_i is not nogood.
3. [Completeness.] Every consistent environment E in which l holds is a superset of some E_i.
4. [Minimality.] No E_i is a proper subset of any other.

All ATMS algorithms maintain an auxiliary set $\mathcal{N}$ of unsubsumed nogoods. For an ideal ATMS $\mathcal{N}$ is exactly the set of minimal nogoods. However, identifying all minimal nogoods is expensive and thus, in practice, ATMS algorithms usually identify only some of the nogoods, thereby (only) foregoing the above consistency property but at great improvement in efficiency. All ATMS algorithms ensure a weakened form of consistency:

2. [Consistency'.] E_i is not subsumed by $\mathcal{N}$.

3 Propositional encoding of CSPs

A CSP is encoded as a set of propositional clauses. We refer to the mapping of constraint networks to a clause sets $\mathcal{C}$ as $\mathcal{C} = \Delta(CN)$. For every x_i we introduce a propositional symbol for every possible value $a_{ij} \in D_i$. We write these propositional symbols as $x{:}a$ which represents the proposition that $x = a$. The constraints are formulated as a set of clauses $\mathcal{C}$ in terms of these propositional symbols as follows. For each variable x_i we include in $\mathcal{C}$ the positive clause (i.e., a choose) which states that every variable must have a value:

$$x_i{:}a_{i1} \vee \cdots \vee x_i{:}a_{in_i}.$$

In addition, we specify that a variable can have at most one value; for every x_i, for every a_{ij}, for every a_{ik} such that $k \neq j$ we include in $\mathcal{C}$ the binary negative clause:

$$\neg x_i{:}a_{ij} \vee \neg x_i{:}a_{ik}.$$

Every constraint C_Z in the constraint network is encoded as follows. For every tuple $[t_1, \ldots, t_m]$ of assignments to the variables of Z we need a formula disallowing that conjunction:

$$\neg(t_1 \wedge \cdots \wedge t_m),$$

which in clausal form is:

$$\neg t_1 \vee \cdots \vee \neg t_m.$$

We have encoded the three conditions which define a constraint problem: every variable has a value, no variable can have more than one value, and no combination of variable assignments violates any constraint. Therefore, every complete assignment of truth values to the propositional symbols which satisfies every clause in C, corresponds to a solution to the original CSP. Conversely, every solution to the CSP corresponds to an assignment of truth values to the propositional symbols which satisfies every clause of *C*.

3.1 Example continued

We were given the variables x_1 x_2,x_3 with domains D_1 = {a, b}, D_2 = {e,/}, and D_3 = {c, *d,g)* and binary constraints C_{12} = {be,bf}, C_{13} = *{bc.bd.bg},* and C_{23} = *{ed,fg}.* *C* consists of the following clause set. Each variable must have a value:

$$x_1{:}a \vee x_1{:}b,\ x_2{:}e \vee x_2{:}f,\ x_3{:}c \vee x_3{:}d \vee x_3{:}g$$

The negative clauses stating that each variable can have only one value:

x_1 :	x_2 :	x_3 :
$\neg x_1{:}a \vee \neg x_1{:}b$	$\neg x_2{:}e \vee \neg x_2{:}f$	$\neg x_3{:}c \vee \neg x_3{:}d$
		$\neg x_3{:}c \vee \neg x_3{:}g$
		$\neg x_3{:}d \vee \neg x_3{:}g$

The constraints:

C_{12} :	C_{13} :	C_{23} :
$\neg x_1{:}a \vee \neg x_2{:}e$	$\neg x_1{:}a \vee \neg x_3{:}c$	$\neg x_2{:}e \vee \neg x_3{:}c$
$\neg x_1{:}a \vee \neg x_2{:}f$	$\neg x_1{:}a \vee \neg x_3{:}d$	$\neg x_2{:}e \vee \neg x_3{:}g$
	$\neg x_1{:}a \vee \neg x_3{:}g$	$\neg x_2{:}f \vee \neg x_3{:}c$
		$\neg x_2{:}f \vee \neg x_3{:}d$

3.2 CSP-Propositional mapping

Every constraint network *CN* has a unique encoding *C* = $\Delta(CN)$ as a set of clauses. The inverse, $CN = \Delta^{-1}(C)$ exists only under the following conditions:

1. All clauses are either positive or negative.
2. Every symbol appears in at most one positive clause.
3. For every pair of symbols in every positive clause, the binary negative clause of these two symbols is in the clause set.
4. No negative clause of three or more literals contains two symbols occurring in the same positive clause.

The following defines Δ^{-1}: (1) every positive clause defines a variable and (2) every negative clause mentioning only symbols which occur in positive clauses but which does not mention two symbols from the same positive clause indicates a disallowed combination for the constraint among the variables of the symbols. Note that for every constraint network CN: $CN = \Delta^{-1}(\Delta(CN))$

3.3 Backtrack search

Both ATMS and CSP techniques exploit backtrack search for identifying global solutions. The following is a simple generic backtrack algorithm for constructing solutions to a CSP encoded as a set of clauses *C*. Let Subsumed?(S, *C)* be a function which checks whether the negative clause constructed from the symbols of *S* is subsumed by *C* (equivalently, the function checks whether interpreting the symbols of *S* to be true falsifies any clause of *C).* Let the global variable *b* count the number of backtracks (i.e., the number of times BT was called without producing a solution), *b* is initialized to be 0.

ALGORITHM BT(S,C):

1. [Termination check] If every variable has a value in S, then mark *S* as a solution and return.
2. [Subsumption check] Pick a variable X_i which does not have a value in *S*. If Subsumed?$(S \cup \{x_i{:}a_{ij}\})$ holds for every value a_{ij}, then set *b* = *b* + 1 and return.
3. [Iterate] For every j, unless Subsumed?$(S \cup \{x_i{:}a_{ij}\})$ call $\mathbf{BT}(S \cup \{x_i{:}a_{ij}\})$.

Note that BT implicitly incorporates the binary negative clauses involving elements of the same variable.

Unfortunately, using BT to find all solutions can involve an enormous amount of often futile [4, 13] backtracking (i.e., 6 will be very large). Both ATMS and CSP algorithms incorporate the strategy which, in propositional terms, amounts to constructing additional clauses, entailed by *C*. Adding to *C* a clause entailed by it, can only reduce the number of backtracks required to find a solution. Adding all clauses to *C* entailed by it, guarantees backtrack-free search.

4 Propositional inference rules

All ATMS and CSP local consistency techniques can be viewed as implementing a few basic resolution rules. All of these inference rules change the clause set, preserving the set of global solutions, but improving the efficiency of the backtrack search used to find the solutions. This section presents a set of inference rules referred to as I which we use in subsequent sections to characterize the ATMS and CSP local consistency algorithms. The names of these rules are chosen for compatibility with the equivalent ATMS rules.

HO: This rule removes subsumed clauses from *C*. If clause $D \in C$ is subsumed by some other clause $E \in C$, then *D* is removed from *C*.

H3: A unit resolution rule:

$$\frac{\neg A \qquad A \vee A_1 \vee \cdots \vee A_m}{A_1 \vee \cdots \vee A_m \text{ replaces the antecedent disjunction}}$$

H5: This rule does the main work. *N(B)* indicates the negative clause with symbols B. Given a set of negative clauses *N(ai):*

$$\frac{A_1 \vee \cdots \vee A_m \qquad N(\alpha_i) \text{ where } A_i \in \alpha_i \text{ and } A_{j \neq i} \notin \alpha_i \text{ for all } i}{N(\bigcup_i [\alpha_i - \{A_i\}])}$$

An instance of this rule is:

$$\frac{x_2{:}e \vee x_2{:}f \qquad \neg x_2{:}e \vee \neg x_3{:}c \qquad \neg x_2{:}f \vee \neg x_3{:}c}{\neg x_3{:}c}$$

H5 can a generate a very large number of clauses, therefore it is useful to define two restrictions of H5.

H5-k:: H5 restricted to only infer negative clauses below size *k*. The basic intuition behind this restriction is that it

reduces overall computational complexity by reducing the number of clauses H5 introduces. In addition, when H5 is used as part of search, smaller clauses are more valuable than larger ones because they provide more pruning. Given a set of negative clauses N(a,):

$$\frac{A_1 \vee \cdots \vee A_m \quad N(\alpha_i) \text{ where } A_i \in \alpha_i \text{ and } A_{j \neq i} \notin \alpha_i \text{ for all } i}{N(\bigcup_i [\alpha_i - \{A_i\}]) \text{ where } |\bigcup_i [\alpha_i - \{A_i\}]| < k}$$

(Note that the rule will only apply to $|\alpha_i| \leq k$.)

5 Local consistency conditions

The local consistency algorithms enforce local consistency conditions by adding clauses. These added clauses reduce the backtracking needed to find solutions.

In discussing these theorems it is important recognize that the constraints in a constraint network, by construction, do not contain spurious information. For example, if a constraint network has the constraint C_{12} = *{ab},* then necessarily $a \in D_1$ and $b \in D_2$. Thus, all CSP local consistency algorithms implicitly incorporate a limited form of subsumption: when an element is removed from a domain of some variable, all constraints referring to that variable are simplified to no longer to refer to the deleted element. (Note that this happens automatically if constraints are represented by allowed combinations, however, how a constraint is represented is not specified in the CSP approach.)

5.1 Node consistency

Node consistency ensures that the singleton constraint *Ci{X)* for every node *i* and any value $X \in D_i$. Node consistency ensures that all unary constraints are used to delete disallowed domain elements. In the clausal encoding, the domain of a variable is represented by a single disjunction and unary constraints appear as singleton negative clauses.

Theorem 1 *A constraint network CN is node consistent iff no application of IIS to* $\mathcal{C} = \Delta(CN)$ *produces a clause.*

5.2 Arc consistency

Arc consistency is ambiguously defined in the CSP literature. Here we adopt the definition of [14] which is slightly stronger than the one in [13]. Note that arc consistency does not necessarily imply node consistency, although all so-called arc consistency algorithms in the literature enforce node consistency as well. Arc consistency ensures that for every pair of nodes *i* and j, for every value $X \in D_i$ there exists a value $Y \in D_j$ which is not excluded by C_{ij}. (To avoid double subscripts, I write *x* for x, and *y* for *Xj.)*

Theorem 2 *A constraint network CN is arc consistent iff no application of H5-2 to* $\mathcal{C} = \Delta(CN)$ *produces a singleton negative clause.*

Proof. Let the Y_1 be the elements of D_j. This is represented by the disjunction:

$$y{:}Y_1 \vee \cdots \vee y{:}Y_{n_y}.$$

Suppose that there were some value $X \in D_i$ such that every value $Y \in D_j$ was excluded by C_{iJ}. These exclusions are represented by binary negative clauses. Thus, there must exist a set of binary negative clauses all of which contain *x:X* and each of the possible values

$$\neg x{:}X \vee \neg y{:}Y_1$$

$$\neg x{:}X \vee \neg y{:}Y_n.$$

Applying H5-2 to these binary clauses we obtain the singleton negative clause *-ix:X.*

The converse is straight forward.

5.3 Path consistency

As with arc consistency, path consistency is also ambiguously defined. We adopt the stronger definition of path consistency from [14]: if for every three nodes r, *s* and / if for every value $X \in D_r$ and $Z \in D_t$ not excluded by the constraint between nodes C_{rt} there exists a value $Y \in D_s$ such that the values *X* and *Y* are not excluded by $C_{re)}$ and the values *Y* and *Z* are not excluded by C_{st}.

Theorem 3 *A constraint network CN is path consistent iff no application of H5-3 to* $\mathcal{C} = \Delta(CN)$ *produces a binary clause not subsumed by C.*

5.4 k-consistency

[10] generalizes the preceding result to obtain:

Theorem 4 *A constraint network CN is strongly k-consistent iff no application of H5-k to* $\mathcal{C} = \Delta(CN)$ *introduces a new clause not subsumed by C.*

6 Local consistency algorithms

All ATMS and CSP local consistency algorithms employ some subset of the inference rules J (from section 4) to modify the clause set *C* to achieve local consistency. Given the theorems of the preceding section and the set of inference rules I, the following useful algorithmic properties hold for a clausal encoding of a CSP.

Proposition 5 *Given any subset of inference rules from I, any order of application will lead to the same resulting clause set as long as the clause set closed under those rules.*

Proposition 6 *Any algorithm incorporating any subset of inference rules from 1 achieves node consistency as long as the resulting clause set is closed under H3.*

Proposition 7 *Any algorithm incorporating any subset of inference rules from 1 achieves node and arc consistency as long as the resulting clause set is closed under H3 and H5-S.*

Proposition 8 *Any algorithm incorporating any subset of inference rules from 1 achieves node, arc and path consistency as long as the resulting clause set is closed under H3 and H5-S.*

Proposition 9 *Any algorithm incorporating any subset of inference rules from 1 achieves strong k-consistency as long as the resulting clause set is closed under H5-k.*

6.1 The importance of the algorithms

The basic motivation for applying the local consistency algorithms to constraint networks is to reduce or even totally eliminate the number of backtracks required to identify the solutions. This issue is extensively discussed in the CSP literature. A general result from Freuder stated in propositional terms is:

Theorem 10 *// the clause set C = A(CN) is closed under H5, then the solutions can be found via backtrack-free search.*

The local consistency conditions are similarly important to ATMS operations — often backtrack search is used to construct problem solutions. The local consistency conditions however play an even more central role in ATMS functioning. Fundamental to the ATMS is its ability to efficiently answer queries whether the conjunction of a set of symbols is known to be inconsistent. If the ATMS replies that the symbol set is consistent when, in actual fact, it is not, the inference engine may perform an unbounded amount of futile work. Therefore, it is important that the ATMS answer such queries correctly. If the clause set is closed under H5, then the ATMS can correctly answer the queries by a simple subsumption test with *N.*

Extensive experience with the ATMS has shown that closing the initial clause set under H5 usually yields a total increase in overall problem-solving effort. This is because the inference engine effort saved by correctly answering the queries is outweighed by the computational effort of closing the clause set under H5. Usually, it is globally optimal to close the clause set under 115-k for some small value of *k.* These conditions, of course, correspond to the ATMS notions of arc and path consistency.

7 CSP algorithms

Both ([13] and [14]) definitions of local consistency admit more ambiguity than first meets the eye. Consider just the simple case of achieving node consistency. A node consistency algorithm converts a constraint network to an equivalent one which is node consistent. Unfortunately, there may be a very large number of equivalent networks which are node consistent. The node consistency condition, although well-defined, does not specify which of these networks a node consistency algorithm must find. Although rarely specified, all consistency algorithms incrementally remove local consistency violations by minimally tightening the constraints to find an equivalent consistent network meeting the local consistency condition.

The common CSP local consistency algorithms can be analyzed in terms of the inference rules of *2.* The node consistency algorithm NC-1 repeatedly applies rule 113.

Theorem 11 *Executing NC-1 to constraint network CN has the same result as applying* A^{n1} *to the closure of A(CN) under H3 and H5-2.*

All arc consistency algorithms close the clause set under H3 and H5-2, applying H3 first whenever possible. The different arc consistency algorithms do various amounts of bookkeeping to prevent futile applications of H5-2.

Theorem 12 *Executing AC-1, AC-2, or ACS to constraint network CN has the same result as applying* A^{*1} *to the closure of A(CN) under H3 and H5-2.*

Analogously, the path consistency algorithms close the clause set under H3 and H5-3.

Theorem 13 *Executing PC-1 or PC-2 to constraint network CN has the same result as applying* A^{*1} *to the closure ofA{CN) under H3 and H5-S.*

8 ATMS algorithms

The ATMS contains a variety of mechanisms to maintain node labels which we do not discuss here. Instead, this paper focuses on ATMS techniques to construct nogoods entailed by *C.* The ATMS represents the clauses as N a set of nogoods (i.e., negative clauses) and *V* a set of chooses (i.e., positive clauses) where every symbol representing an assignment is an ATMS assumption. As the ATMS explicitly represents clauses it can directly incorporate the inference rules I. All versions of the ATMS algorithms incorporate some subset of these inference rules.

8.1 Efficiency considerations

All ATMS algorithms incorporate the subsumption rule HO. The motivation for incorporating HO is that there are an exponential number of potential clauses to construct, but, on average, the subsumption rule will eliminate most of them from the clause set. Therefore, in all ATMS algorithms HO is always performed first.

Property 5 of section 4 guarantees that the application order of inference rules is irrelevant. Therefore ATMS implementations order the application of inference rules to minimize total effort. The basic intuitions behind the ordering heuristics are that H5 is extremely expensive and should always be applied last and to smaller nogoods first — smaller nogoods should be discovered first as they subsume far more higher-order nogoods which the algorithm might futilely have to process.

It should be noted that neither rules HO or H3 are necessary to ensure backtrack free search. Both are included for efficiency considerations. Note that the use of rule HO makes the overall algorithm adaptive - it allows an ATMS algorithm to dynamically adapt the set of nogoods to resolve as it eliminates nogoods from consideration.

8.2 ATMS algorithm

Whenever a nogood becomes subsumed, as well as being removed from N it is removed from all pending queues, and all processing on it is halted. The algorithm maintains a queue *L* of pending nogoods ordered by size, initially *L* is all new nogoods of size *k* or less. In usual practice RESOLVE is invoked after every atomic ATMS operation.

ALGORITHM RESOLVE:

1. [Unit preference for H5] If there is a singleton choose, its single literal is set to true, and every nogood *ng* mentioning the literal is replaced by a new nogood with that literal deleted.
2. If *L* is empty, return. Pick and remove *ng* a smallest nogood from *L.* If *L* is empty, return.
3. If *ng* is the empty nogood, halt. The constraints are unsatisfiable.
4. [H3] If *ng* is a singleton, it is removed from the choose(s) in which it appears. All processed un-

subsumed nogoods which mention elements in these chooses are requeued on L. Go to step 1.

5. [H5] For every element e of ng, find its choose(s) d, and for each element of d collect all the processed unsubsumed nogoods in which it appears. Iteratively, pick one of these nogoods for each element of d, union their elements, and delete the elements of d. Each new set thus constructed is a new nogood $ng*$. If this $ng*$ is subsumed, do nothing. Otherwise, insert it into the nogood data-base and in L if it is of size less than k.
6. Go to step 1.

Although this algorithm is designed to identify smaller nogoods first, it cannot guarantee this. For example, while processing nogoods of size four, it might produce a new nogood of size three. Note that organizing the search to process smaller nogoods first is equivalent to organizing the search to process smaller constraints first. Thus we can guarantee that if the above algorithm has ever processed a nogood of size m, that it has processed every constraint of size $l < m$.

The following is the execution trace of the first few iterations of the algorithm applied to our example (with $k = n$). Initially, both the nogood data-base and L consist of the following. For clarity the binary nogoods involving elements of the same variable are left out. Chooses and nogoods are printed out as sets of symbols.

Initial nogoods: $\{\{x_2{:}e, x_3{:}c\}, \{x_2{:}f, x_3{:}c\}, \{x_2{:}f, x_3{:}d\}, \{x_2{:}e, x_3{:}g\}, \{x_1{:}a, x_3{:}c\}, \{x_1{:}a, x_3{:}d\}, \{x_1{:}a, x_3{:}g\}, \{x_1{:}a, x_2{:}e\}, \{x_1{:}a, x_2{:}f\}\}$

Picking: $\{x_2{:}e, x_3{:}c\}$

Picking: $\{x_2{:}f, x_3{:}c\}$

Resolving the nogoods $\{x_2{:}e, x_3{:}c\}$ and $\{x_2{:}f, x_3{:}c\}$ with choose $\{x_2{:}e, x_2{:}f\}$ gives the new nogood $\{x_3{:}c\}$.

Inserting $\{x_3{:}c\}$ into the new the nogood data base removes the subsumed, $\{\{x_2{:}e, x_3{:}c\}, \{x_2{:}f, x_3{:}c\}, \{x_1{:}a, x_3{:}c\}\}$, leaving the nogoods, $\{\{x_3{:}c\}, \{x_2{:}f, x_3{:}d\}, \{x_2{:}e, x_3{:}g\}, \{x_1{:}a, x_3{:}d\}, \{x_1{:}a, x_3{:}g\}, \{x_1{:}a, x_2{:}e\}, \{x_1{:}a, x_2{:}f\}\}$.

Picking: $\{x_3{:}c\}$

Eliminating it from the choose: $\{x_3{:}c, x_3{:}d, x_3{:}g\}$.

9 Comparison of algorithms

Due to the fact the ATMS always performs subsumption, different constraint networks may be represented by the same $\mathcal{N}$ and $\mathcal{D}$. Suppose constraint C_{123} disallowed the combination abc and constraint C_{12} disallowed the combination ab. After encoding the constraints as nogoods, the ATMS throws away the nogood abc because it is subsumed by ab. Hence the same $\mathcal{D}$ and $\mathcal{N}$ corresponds to (at least) two equivalent constraint networks — one in which C_{123} allows abc and another which does not. To avoid these difficulties we define the following:

Definition 14 *A constraint network is irredundant if every tuple T of assignments disallowed by every constraint C_Z is not disallowed by some other constraint C_Y where $Y \subset Z$.*

Definition 15 *The irredundant form of a constraint network is an equivalent irredundant network formed by repeatedly removing excluded tuples in higher-order constraints also excluded by lower-order constraints.*

Given clause set C we can always construct an equivalent irredundant constraint network as follows. If a clause set C is closed under H0, then $CN = \Delta^{-1}(C)$ is irredundant.

9.1 Comparison of results

As a consequence of propositions 6, 7 and 8 (see section 4) the constraint network represented by (i.e., via Δ^{-1}) a set of clauses is node, arc and path consistent after the ATMS algorithm has been executed for $k = 1, 2, 3$ respectively.

The standard ATMS algorithms always apply H5 to unit chooses immediately. This unit preference strategy is, in my experience, always worth it. However, the CSP algorithms do not incorporate this strategy for H5 (even though they incorporate, as the ATMS does, the unit resolution rule H3). As a consequence of the ATMS's unit preference strategy, the irredundant form of the constraint network produced by a CSP algorithm is not necessarily the same as the one produced (via Δ^{-1}) by an ATMS algorithm. For example, running the standard node consistency algorithm NC leaves the network $D_x = \{A\}, D_y = \{B, C\}, C_{xy} = \{A, C\}$ unchanged because it is already node consistent. However, after processing with $k = 1$ the ATMS has reduced the network to $D_x = \{A\}, D_y = \{C\}$. However, we could delete step 1 from **RESOLVE** (step 5 would deduce the same nogood later anyway). So modified, the irredundant form of the constraint network produced by the standard node and arc consistency algorithms is the same (via Δ^{-1}) as the ATMS produces with $k = 1$ and 2 respectively.

The comparison with path consistency is more difficult. Instead of dwelling on path consistency, it is more useful to consider the generalized notion of k-consistency. The irredundant form of the network produced by Freuder's synthesis algorithm [11] is the same (via Δ^{-1}) as **RESOLVE** with step 1 disabled.

See [10] for a comparison of the computational complexities.

10 Comparison of the approaches

In this paper we have focused on ATMS concerns which relate to CSPs. However, an ATMS is a very general problem solver facility which provides far more functionality than any CSP algorithm. For example, constructing explanations, doing differential diagnosis, interleaving formulation and solving, etc. As a consequence, it is unlikely that an implementation of a constraint solver using the ATMS would use much of the encoding of 3.1. The CSP framework is not concerned with where the constraints come from nor with the cost of evaluating constraint predicates on assignment tuples. The CSP framework also does not permit dynamic reconstruction of the constraint problem. These, among others, are important problem solving issues. Thus, ATMS usage is oriented towards adaptively minimizing the number of constraints constructed, minimizing predicate evaluations, and minimizing variable domain sizes.

10.1 Limiting predicate evaluations

Suppose that the problem at hand is a fixed CSP, except that it is extremely expensive to determine whether a set of values satisfy some constraint predicate C. For simplicity we assume that all such determinations are equally costly.

Consider the CSP discussed earlier: Variables x_1, x_2, x_3 with domains $D_1 = \{a, b\}$, $D_2 = \{e, f\}$, and $D_3 = \{c, d, g\}$ and binary constraints $C_{12} = \{be, bf\}$, $C_{13} = \{bc, bd, bg\}$, and $C_{23} = \{ed, fg\}$. Encoding the problem as a CSP requires evaluating every predicate on every combination of values. Thus C_{12} requires 4 evaluations, C_{13} requires 6 evaluations, and C_{23} requires 6 as well. Suppose that the ATMS-based problem solver searches for a solution by first instantiating C_{23}, then C_{13}, and then C_{12}. This corresponds to the ATMS execution trace presented in section 8.2. None of 6 evaluations of C_{13} can be avoided. However, after evaluating C_{13}, c is determined to be inconsistent, therefore constraint C_{13} need not be evaluated on ac or bc (in general if any combination of values has been proven inconsistent by previous constraints it need not be tried). After doing the 4 remaining evaluations of C_{13}, a is determined to be inconsistent. Hence, C_{12} need only be evaluated on be and bf. Thus, the ATMS has saved 4 of the 16 expensive predicate evaluations. For more complex CSP tasks, for example if there were thousands of constraints mentioning x_1, x_2, and combinations of other variables, the ATMS strategy yields far greater performance improvements. One could imagine a lazy CSP which evaluated predicates as needed, but unless some technique (such as perhaps adaptive consistency) is used to control the constraints examined, not enough would be gained.

So far (as in section 3.3) we have been viewing backtracking as a post-processing technique to be invoked after the constraints (or clauses) have been processed by a local consistency algorithm. Particularly in the ATMS framework, local consistency calculations are easily intermingled with backtracking. In fact, backtracking can be used as a control technique to limit unnecessary evaluation of constraint predicates [7]. The combination of backtracking and the ATMS resolution rules produces as good a dependency-directed backtracking scheme as is possible (given the presumption that no additional reasoning over unevaluated predicates is permitted, i.e., given no additional information other than the clauses which the ATMS has been provided by the inference engine). Most ATMS solvers for CSP-like problems will incorporate some form of backtracking to control the evaluation of predicates. This is discussed in detail in [7].

10.2 Constraint propagation

CSP algorithms typically represent constraints by the extension of the allowed or disallowed combinations of values. Most constraints cannot be efficiently encoded this way. Suppose that we were given the constraint $x = y + 1$ and the fact that x and y had 10000 values each. In this case it is better not to encode both x and y as ATMS chooses as the encoding of section 3.1 would. Neither would it be reasonable to encode the constraint in terms of its allowed combinations. Rather, only y should be encoded as a variable, and problem-solver rules should be attached to it such that if a value were obtained for it a value is computed for x as well. This idea is called constraint propagation. In the ATMS framework the values for x would be encoded as a set of non-assumption nodes and as the search encountered y's values, justifications would be installed for the nodes of x. Most constraints and constraint variables should be encoded this way instead of as described in section 3.1. This approach maximizes the use of ATMS justification-handling facilities which are efficient, minimizes the number of assumptions necessary to encode a CSP, and minimizes the use of the relatively inefficient hyperresolution rules.

If constraints are properly [6] encoded as justifications and problem-solving rules, then neither soundness nor completeness is lost. A great deal of efficiency is gained and the reasoning that underlies the construction of the constraints is explicit and under the control of the problem solver. Taking a bird'seye view of constraint propagation applied to CSPs we recognize that, it transforms a CSP into a smaller and more tractable one but doing so under explicit problem solving control such that search and predicate evaluation is minimized.

11 Acknowledgments

This paper profited greatly from lengthy discussions with John Lamping, Alan Mackworth and Ramin Zabih. Also, Daniel G. Bobrow, Rina Dechter, Mike Dixon, Ken Forbus, Felix Frayman, Ken Kahn, Sanjay Mittal, Judea Pearl, Raymond Reiter, Jeffrey Siskind, Mark Stefik, and Brian Williams provided valuable comments on this paper.

References

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

[2] de Kleer, J., An assumption-based truth maintenance system, *Artificial Intelligence* 28 (1986) 127-162. Also in *Readings in NonMonotonic Reasoning,* edited by Matthew L. Ginsberg, (Morgan Kaufman, 1987), 280-297.

[3] de Kleer, J, Extending the ATMS, *Artificial Intelligence* 28 (1986) 163-196.

[4] de Kleer, J., Problem solving with the ATMS, *Artificial Intelligence* 28 (1986) 197-224.

[5] de Kleer, J. and Williams, B.C., Back to backtracking: Controlling the ATMS, *Proceedings of the National Conference on Artificial Intelligence,* Philadelphia, PA (August 1986), 910-917.

[6] de Kleer, J., Propositional inference in CSP and ATMS techniques, SSL Paper P89-00023, 1989.

[7] Freuder, E.C., Synthesizing constraint expressions, *Communications of the ACM* 21(11) (1978) 958 966.

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

[9] Mackworth, A.K., Constraint satisfaction, *Encyclopedia of Artificial Intelligence,* edited by S.C. Shapiro, (John Wiley and Son, 1987) 205-211.

[10] Reiter, R. and de Kleer, J., Foundations of assumption-based truth maintenance systems: preliminary report, *Proceedings of the National Conference on Artificial Intelligence,* Seattle, WA (July, 1987), 183-188.