NoGood Caching for MultiAgent Backtrack Search

William S. Havens
Intelligent Systems Laboratory
School of Computing Science
Simon Fraser University
Burnaby, B.C., Canada V5A 1S6
e-mail: havens@cs.sfu.ca

(Research Paper)

Abstract
Multiagent solutions to the distributed constraint satisfaction problem (DCSP) require new types of techniques which accommodate the local autonomy of agents and the difficulties of computing in a network environment. Recently a technique called asynchronous backtracking has been developed to solve the DCSP. The algorithm works by sending nogood messages among agents to effect intelligent backtracking. One important issue is developing nogood caching schemes which are appropriate to asynchronous backtracking. There has also been recent progress in a sequential algorithm called dynamic backtracking which exhibits a polynomial bound on nogood cache size. In this paper, we show by example that the existing caching scheme used by dynamic backtracking is not appropriate for the multiagent context. We suggest two alternate nogood caching schemes and two caching algorithms based on these schemes. Experimental comparisons of these caching algorithms are forthcoming.

Introduction

Distributed constraint satisfaction problems (DCSPs) are an appropriate abstraction for multiagent cooperative problem solving. They are characterized by multiple reasoning agents making local independent decisions about a global common constraint satisfaction problem (CSP). The motivation for studying the DCSP is not solely speedup of sequential algorithms but the wider issue of coordinating search among multiple and perhaps disparate agents. The applications of multiagent systems in network environments are proliferating but generally focus on straightforward negotiation issues. The next step in this evolution will be multiple agents who collaborate to solve combinatorial tasks as DCSPs. Thus, an important research focus is the development of effective distributed constraint solving algorithms for DCSPs.

The recent work by [Yokoo et al. 92] in adapting intelligent backtracking (IB) techniques to solving DCSPs has generated considerable interest. The method, called asynchronous backtracking (AB), constructs inconsistent sets of variable bindings (nogoods) and communicates them among multiple agents to effect an IB search. The method has also been extended to incorporate heuristic search [Yokoo 93]. The key idea is that agents can make local autonomous decisions yet effect a global systematic search.

Since AB is claimed to be complete, an obvious question to ask is: How space efficient is the algorithm? In particular, how many nogoods are derived and then communicated among agents during the search? The reports cited above do not indicate the size of the nogood store actually computed. But its size is certainly crucial to practical application (especially across slow communication media such as the internet). We seek to understand this issue.

Recent advances have also been made in sequential IB algorithms. A technique called dynamic backtracking (DB) [Ginsberg 93] is a form of IB which provides considerable flexibility in the search direction while maintaining completeness. Furthermore, the algorithm exhibits a polynomial space bound of O(n^2d) on the number of nogoods stored at any time [Ginsberg&McAllester 94].

In this paper, we analyse the nogood caching behaviour of DB from the perspective of its application to the DCSP. In particular, can a similar polynomial space bound be established for AB which also supports the autonomy of the multiagent approach? The answer is a qualified yes. We show that the existing caching scheme defined for DB is inadequate but can be modified to the advantage of DCSP search.

In the next section, we introduce the DCSP and the basic method of intelligent backtracking. From this framework, we then consider asynchronous backtracking and dynamic backtracking. In particular, the nogood caching rule given by Ginsberg and McAllester [94] is examined. In the following section, a DCSP example is given which is problematic for the DB caching rule. It is already known that this caching rule preemptively discards nogoods which must be recomputed repeatedly [Baker 95]. For AB, this is a particularly inefficient scheme because it unnecessarily resets the states of agents not involved in the backtracking. The problem is
exacerbated in the network environment where we seek to minimize communication between agents.

Next we consider an alternative caching rule which removes this difficulty. It is shown that our new caching rule supports a variety of caching algorithms. Two such algorithms are proposed. This is a preliminary report on nogood caching which has application in both the CSP and the DCSP. Experimental confirmation of these results are forthcoming.

**Distributed CSPs and Backtrack Search**

**Definition 1:** A constraint satisfaction problem, CSP = \( (X_N, C) \), where \( X_N \) is a set of discrete variables and \( C \) is a set of \( k \)-ary constraints on these variables.

We will refer to subsets of the variables in \( X_N \) using index set notation. The set of all subscripts is \( \mathbb{N} = \{1, \ldots, n\} \). The complete set of \( n \) variables in the CSP is then \( X_N \). A particular subset of variables \( X_J \subseteq X_N \) refers to the variables \( \{X_j\}_{j \in J} \) such that \( J \subseteq \mathbb{N} \). Each variable \( X_i \in X_N \) has a domain \( D_i \) of discrete values. Likewise any constraint \( C_j \in C \) is a relation on variables \( X_J \subseteq X_N \).

**Definition 2:** A distributed constraint satisfaction problem (DCSP) is a CSP where the variables \( X_N \) and the constraints \( C \) are distributed among a finite set of agents.

There are many possible mappings of variables and constraints to agents. For simplicity, we assume (like AB) that each agent has exactly one variable and knows every constraint on that variable. Unlike AB, constraints are not binary directed arcs in the constraint graph but general \( k \)-ary predicates \( C_k \) for \( k = |I| \). We will informally substitute variables for agents and vice versa when the distinction is not important.

**Definition 3:** The binding of a variable \( X_i \) to some element \( \alpha \) is \( D_i = \alpha \).

We can abbreviate the notation by referring simply to \( \alpha \) as the binding since the subscript "i" identifies which variable \( X_i \) it is assigned. A set of bindings (a tuple) is denoted also using index set notation. Hence \( \alpha_J = \{\alpha_j\}_{j \in J} \) represents the set of bindings for variables \( X_J, j \in J \). A constraint \( C_j \) holds on this tuple iff \( \alpha_j \in C_j \).

**Definition 4:** A solution to the DCSP is a tuple \( \alpha_N \) for all variables \( X_N \) such that \( \alpha_J \subseteq C_J, \alpha_J \subseteq X_N \), which must hold for every constraint \( C_j \in C \).

Since \( \alpha_N \) contains a binding for every variable in the CSP, we can think of it as encoding a point in the search space. As the search moves in this space (or some subspace), individual constraints are variously consistent or inconsistent. Whenever a constraint becomes inconsistent, a record of the inconsistency is made in the form of a nogood.

**Definition 5:** The negation of an inconsistent tuple, written \( \neg \alpha_J \), is called a nogood. This assignment of values to variables \( X_j \) is known not to satisfy some constraint(s) in \( C \).

Nogoods are constructed either by the failure of a given constraint or by resolution from existing nogoods. If the empty nogood is derived (containing no bindings) then the CSP is unsatisfiable. For details, see for example [Ginsberg & McAllester 94].

**Definition 6:** (culprit selection rule) In any nonempty nogood there is a distinguished binding called the culprit which is the most recent binding made in the ordering of variables. In the nogood \( \neg \alpha_J = \{\alpha_j\}_{j \in J} \), the culprit is binding \( \alpha_t \in \alpha_J \) such that \( t \geq j \) for all \( j \in J \).

**Definition 7:** A nogood expressed in culprit form is \( \neg \alpha_J = \neg(\alpha_K \alpha_i) \), where \( \alpha_i \) is the culprit binding according to the culprit selection rule and \( \alpha_K \), \( K=1-\{t\} \), is the antecedent.

A nogood in culprit form can also be expressed equivalently as the implication \((\alpha_K \Rightarrow \neg \alpha_i)\) such as used by [McAllester & Ginsberg, 94]. In general, let \( \Gamma_i \) be the nogood cache for agent "i" containing the set of nogoods derived for culprit variable \( X_i \). The global nogood cache \( \Gamma \) is simply the set of all the agent’s local stores, \( \Gamma = \{\Gamma_i\}_{i \in \mathbb{N}} \).

**Definition 8:** The subset \( \Delta_i \subseteq D_i \) is called the live domain of variable \( X_i \) which represents those values in its domain \( D_i \) which are consistent with the current partial solution.

The live domain for \( X_i \) is then those values \( \alpha_i \in D_i \) which are not precluded by any existing nogood whose antecedent is currently contained in the global point \( \alpha_N \). More precisely:

\[
\Delta_i = \{\alpha_i \in D_i | \forall \neg(\alpha_K \alpha_i) \in \Gamma_i, \alpha_K \not\subseteq \alpha_N\}
\]  

Initially since there are no nogoods, every value in the domain is also in the live domain. Note that the live domain can change radically as the search moves points \( \alpha_i \) in the search space. The values of nogood antecedents \( \alpha_K \) are only true when \( \alpha_K \subseteq \alpha_N \). Our challenge is to manage the nogood cache (and hence the live domains) efficiently as the search proceeds.

**Definition 9:** A variable \( X_i \) whose live domain \( \Delta_i = \{\} \) is called a bottom variable and written \( X_i = \perp \).

**Asynchronous versus Dynamic Backtracking**

First we review the method of asynchronous backtracking from the perspective of nogood generation and propagation. Then we proceed to analyse the application of dynamic backtracking within this framework.

**Asynchronous Backtracking**

Yokoo [92] defines a message passing protocol for realizing IB in the multiagent context. Agents correspond to individual variables which are initially assigned a total order.
Constraints are binary directed links between agents (ordered from lower to higher agents in the order). Agents autonomously choose values for their variables which are then communicated to neighbour (receiving) agents along the directed links. The sending agent sends its value to the receiving agent via an “OK?” message. Receiving agents maintain a tuple of received variable bindings (called its agent-view) which are checked along with its own binding against its constraints.

If all constraints are satisfied then the global set of bindings for the variables constitutes a solution to the DCSP. Otherwise, the agent-view comprises a nogood antecedent which eliminates the current value (the culprit) from the agent’s live domain. The agent then freely (heuristically) chooses a new value from its live domain and recursively sends the OK? message to all connected agents lower in the order. However, if the agent variable reaches bottom by exhausting its live domain, then the agent-view comprises a nogood which is communicated (via a “nogood” message) to the highest variable appearing in the nogood (the culprit selection rule).

When an agent receives a nogood message, first it checks its binding in the nogood. If that value is its current binding then it removes it from its live domain and chooses a new value (if possible) as above. If the empty nogood is derived then the DCSP is unsatisfiable and all agents stop.

Although there are precautions necessary to accommodate the multiprocessing nature of the AB algorithm, basically it is very similar to other forms of IB, in particular DB. It would seem natural to adapt the nogood caching scheme of DB to AB if possible.

Dynamic Backtracking

A major research goal of DB is the maximum “freedom of movement” in the search space while maintaining a systematic and complete search. The authors argue that DB is a balance between these two conflicting criteria. In particular, the caching scheme employed by DB is based on the notion of an “acceptable assignment” (hereafter the AA-rule) which limits the size of the nogood database as the search proceeds through the search space. However, there is a direct conflict between freedom of motion in the search and retention of nogoods. For distributed algorithms, we argue that this conflict is debilitating.

Local autonomy is paramount in the multiagent approach. Each agent must be free to maximize its own utility according to its own heuristics. Interactions among agents should be minimized (in the network environment) and mediated solely by their mutual constraints (and not by details of the nogood store). We assume a precedence order among agents (as per IB and AB). Agents higher in the order must accede to the variable values established by previous agents in the order. In other words, higher agents must backtrack in the environment established by lower agents. Otherwise, agents maximize their local utilities as they are able.

Unfortunately, the DB method cannot achieve this goal in the multiagent context. In particular, the AA-rule forces agents to repeatedly discard their (hard earned) nogood stores as the environment of other agents changes. To make this situation clear, we introduce the nogood caching rule given for DB.

NoGood Caching

We begin by stating the AA-rule in our framework.

Definition 10: (AA-rule) an acceptable assignment is a point $\alpha_N$ that encodes every antecedent in $\Gamma$ and none of the culprits [Ginsberg & McAllester 1994].

More precisely,

$$\forall \neg(\alpha_K \alpha_t) \in \Gamma, \ \alpha_K \subseteq \alpha_N \text{ and } \alpha_t \notin \alpha_N \quad \text{(2)}$$

which says that every nogood, $\neg(\alpha_K \alpha_t)$, in the store must obey the following: 1) the antecedent $\alpha_K$ must be currently true (contained in the current global point, $\alpha_N$) and; 2) $\alpha_N$ cannot contain any binding $\alpha_t$ precluded by such a nogood. The second condition makes sense because it eliminates any point $\alpha_N$ in the search space which is known to be inconsistent given the other bindings $\alpha_K$ contained in this same point.

The first condition is more problematic. It says that any nogood $\alpha_K$ whose antecedent is currently false (i.e. $\alpha_K \not\subseteq \alpha_N$) must be deleted from $\Gamma$. This is the basis for the polynomial space bound on the nogood store. The rule has also been called “l-relevance learning” since the cache contains nogoods which differ from $\alpha_N$ in at most one binding (i.e. the culprit) [Bayardo & Miranker 96].

Algorithm 1: DB version of procedure Cache

Algorithm 1 above gives the nogood caching procedure for DB. This procedure is called whenever a new nogood, $\neg\alpha_N$, is discovered during the search. Cache checks for the empty nogood (in which case the CSP is unsatisfiable) and

otherwise adds \(\neg \alpha_i\) to \(\Gamma\). Let the nogood have an antecedent \(\alpha_F\) and culprit \(\alpha_c\). Then the procedure deletes from \(\Gamma\) every other nogood \(\neg (\alpha_F \alpha_c)\) which contains culprit \(\alpha_c\) in its antecedent \(\alpha_F\). Finally, if this additional nogood forces variable \(X_i = \bot\), then a new nogood \(\alpha^{\bot}\) is induced from the antecedents \(\alpha_F\) of every nogood on \(X_i\) and Cache called recursively on this new nogood.

**A Problematic Example for DB Cache**

How will this procedure Cache based on the AA-rule effect the efficiency of the AB search? Please consider the following example shown in Figure 1.

![Figure 1: A Problematic Example for DB Cache](image)

We have a DCSP with four agents (variables) and two constraints. Constraint \(C_{13}\) relates variables \(X_1\) and \(X_3\) while constraint \(C_{012}\) relates variables \(X_0\), \(X_1\) and \(X_2\). Attached to each variable is the corresponding nogood store \(\Gamma\). Each bullet in the store represents some value \(\alpha\) from its domain and the arrows indicates some existing nogood whose antecedent has eliminated that value from its live domain. For example, variable \(X_1\) has four domain values, two of which have been eliminated by nogoods whose singleton antecedents are the binding \(\alpha_0\) for variable \(X_0\).

For illustration, we assume that the agents for \(X_0\) and \(X_1\) have communicated their bindings (via OK? messages) to the agents for variables \(X_2\) and \(X_3\). Thereafter however, constraints \(C_{012}\) and \(C_{13}\) repeatedly fail on these bindings generating new nogoods of the form \(\neg (\alpha_{01} \alpha_2)\) and \(\neg (\alpha_1 \alpha_3)\) respectively. Successive domain values for \(X_2\) and \(X_3\) are thus ruled out as illustrated. Suppose eventually \(X_2 = \bot\) since its live domain has been completely eliminated through a new nogood \(\neg (\alpha_{03} \alpha_{1})\). The AA-rule then forces the removal of every nogood with culprit \(\alpha_i\) in its antecedent. Thus, neither \(X_2\) nor \(X_3\) will have any nogoods remaining in \(\Gamma\) and their live domains will be fully restored.

The assumption here is that discarded nogoods can be derived again later if required. This is the basis of the \(O[n^2d]\) space bound for \(\Gamma\). But these discarded nogoods will be rediscovered and then discarded again an exponential number of times in the worst case [Baker 95]. Another problem is that some nogoods will never be discovered. Suppose always \(X_2\) reaches bottom before \(X_3\) (e.g. \(|D_2| << |D_3|\) then never will any nogoods be induced from \(X_2 = \bot\). This is because the AA-rule will "reset" the work done by \(X_3\) each time \(X_2\) cycles its live domain asserting the new nogood \(\neg (\alpha_0 \alpha_i)\).

In order to show that the AA-rule unnecessarily discards and then rediscover nogoods, we need the following definition (adapted from [Bayardo&Miranker 96]).

**Definition 11:** The *defining set for* \(X_i\) *is the set of variables* \(X_K\) previous in the variable ordering (i.e. \(\forall k \in K, k < i\)) which can appear in the antecedent of some nogood for \(X_i\).

Intuitively, the defining set \(X_K\) are those variables which when backtracked will cause the nogoods computed for \(X_i\) to be discarded since these nogoods contain antecedent bindings from the current values of these variables \(X_K\). In Figure 2, suppose another variable \(X_j\) shares (part of) the same defining set with \(X_i\). Then when \(X_j\) backtracks into \(X_K\) every nogood in \(X_i\) will be discarded (and vice versa) in the worst case. This is clearly an undesirable caching scheme for any multiagent algorithm such as AB. We would prefer a scheme which left the set of nogoods for \(X_i\) intact as much as possible when \(X_j\) backtracks but still does not require exponential space for the nogood store.

![Figure 2: Nogood interactions between agents](image)

**Improved NoGood Caching Schemes**

In this section, we offer two alternate nogood caching schemes and then suggest two caching algorithms based on these schemes. The first has an unrestricted cache size while the second imposes the same polynomial limit as the AA-rule but without the difficulties noted by example above.

The basic idea of the first scheme is to keep nogoods in the cache whose antecedents are not necessarily contained in the current state (thus violating the AA-rule) but still delete unnecessary nogoods whenever possible. We will allow multiple nogoods to have the same culprit but different antecedents (i.e. \((\alpha_{01}, \alpha_1)\) and \((\beta_{01}, \alpha_1)\)). More precisely, our first improved caching rule is the following:
∀ \neg(\alpha_K \alpha_t) \in \Gamma, (\alpha_K \subseteq \alpha_N \Rightarrow \alpha_t \in \alpha_N) \text{ and } X_t \neq \bot \tag{3}

which says that for every nogood \(\neg(\alpha_K \alpha_t)\) in \(\Gamma\), the current point \(\alpha_N\) must not contain this culprit \(\alpha_t\) if the antecedent \(\alpha_K\) is contained in the current point \(\alpha_N\) and the culprit variable \(X_t\) must not be bottom. So \(\Gamma\) may contain nogoods whose antecedents are not currently true but every such nogood must refer to a culprit which is not currently a bottom variable. The caching strategy implied from this rule is to keep a cache for each variable which contains nogoods whose antecedents range over the entire defining set for the variable but to delete that cache whenever the variable reaches bottom. In particular, we note that the only reason for keeping the antecedents for each nogood whenever the variable is bottom. Thereafter, we can discard these nogoods and thus reset the live domain for the variable.

Given this caching rule, a variety of actual caching algorithms are possible. The simplest version is given in Algorithm-2 below which has no bound on its cache size but exhibits the desired property that backtracking a culprit variable will not discard the caches of other variables which share some defining set with that variable.

\text{Cache}(\neg \alpha_j) \equiv
\begin{align*}
& \text{If } \alpha_j = \emptyset \text{ then Halt.} \\
& \Gamma \leftarrow \Gamma \cup (\neg \alpha_j) \\
& \text{let } (\alpha_H \alpha_L) = \alpha_j \\
& \text{if } X_t = \bot \text{ then} \\
& \quad \text{let } \alpha^L = (\alpha_H \mid (\alpha_H \alpha_L) \in \Gamma \text{ and } \alpha_H \subseteq \alpha_N) \\
& \quad \text{for every } \neg(\alpha_H \alpha_L) \in \Gamma, \\
& \quad \Gamma \leftarrow \Gamma \setminus (\neg(\alpha_H \alpha_L)) \\
& \text{Cache}(\neg \alpha^L) \\
& \text{end.}
\end{align*}

Algorithm 2: Improved procedure Cache with unrestricted space bound.

This version of Cache, like the previous version, first checks for the empty nogood. Otherwise, the new nogood \(\neg \alpha_j\) is added to \(\Gamma\). Let the nogood have an antecedent \(\alpha_K\) and culprit \(\alpha_t\). If variable \(X_t\) is now bottom then perform the following. Resolve a new nogood \(\alpha^L\) from those nogoods in \(\Gamma\) whose culprit is \(\alpha_t\) and whose antecedent is currently true (i.e. contained in the current point). Then delete every nogood from \(\Gamma\) which has culprit \(\alpha_t\) (i.e. empty \(\Gamma_t\)). Finally, recursively cache this new nogood \(\alpha^L\).

Algorithm-2 differs from Algorithm-1 in its strategy for emptying the cache. Instead of discarding nogoods whose antecedents are not contained in the current point (the AA-rule), Algorithm-2 discards nogoods only when their culprits become bottom variables. These nogoods have served their useful purpose and been resolved into a new nogood. They can now be deleted.

Let's consider the example of Figure-1 again using this new caching algorithm. Again constraints \(C_{1012}\) and \(C_{13}\) repeatedly fail generating new nogoods of form \((\alpha_{101} \alpha_2)\) and \((\alpha_1 \alpha_2)\) respectively. Successive bindings for \(X_2\) and \(X_3\) are ruled out. Eventually \(X_2 = \bot\) inducing a new nogood \((\alpha_0 \alpha_1)\). The new Cache algorithm removes only nogoods for \(X_2\) (since \(X_2\) is bottom) while the existing nogoods for \(X_3\) remain intact. Any subsequent nogoods derived for \(X_2\) and \(X_3\) will be of the form \((\beta_{10} \alpha_2)\) and \((\beta_1 \alpha_3)\) respectively (since the binding now of \(X_1\) has necessarily changed). We note that the nogoods are now (possibly) disjunctive for each culprit. Suppose eventually \(X_3 = \bot\) for some antecedent (say \(\beta_1\)) and a new nogood \(((\{\{\beta_1\})\) is derived which means that the algorithm will conclude that the current value \(\beta_1\) for \(X_1\) is globally inconsistent (since the antecedent is null).

Algorithm-2 is an improvement over the original DB version of Cache since it achieves our goal of maintaining autonomy among agents in \(AB\). Nogood caches are not discarded unnecessarily. Communication among agents not involved directly in the backtracking is reduced. However, we have also lost the desirable polynomial space bound provided by DB. Can this bound be reclaimed? We note that there are a wide variety of caching schemes in use (e.g. virtual memory, disk caches). Most of these schemes do not preemptively empty their caches (like the AA-rule) before the cache is actually full! To do so is to unnecessarily discard the results stored therein.

\text{Cache}(\neg \alpha_j) \equiv
\begin{align*}
& \text{If } \alpha_j = \emptyset \text{ then Halt.} \\
& \Gamma \leftarrow \Gamma \cup (\neg \alpha_j) \\
& \text{let } (\alpha_H \alpha_L) = \alpha_j \\
& \text{if } |\Gamma| > \text{Max} \text{ then} \\
& \quad \text{for every } \neg(\alpha_H \alpha_L) \in \Gamma \text{ such that} \\
& \quad \quad \alpha_L \in \alpha_H, \\
& \quad \Gamma \leftarrow \Gamma \setminus (\neg(\alpha_H \alpha_L)) \\
& \quad \text{if } X_t = \bot \text{ then} \\
& \quad \quad \text{let } \alpha^L = (\alpha_H \mid (\alpha_H \alpha_L) \in \Gamma \text{ and } \alpha_H \subseteq \alpha_N) \\
& \quad \quad \text{Cache}(\neg \alpha^L) \\
& \text{end.}
\end{align*}

Algorithm 3: Improved procedure Cache with \(O[n^2d]\) space bound.

From this observation, we offer a second nogood caching scheme by relaxing the requirement that no culprit variable of any nogood can be bottom. Instead we will impose an external constraint on the size of the nogood store. Algorithm-3 above looks very similar to Algorithm-1 except that it refrains from preemptively emptying the cache until the size of \(\Gamma\) is greater than some predetermined value Max. Otherwise, nogoods are retained in the cache in an unrestricted fashion. The value of Max could be set by how much
memory is available or on some other external criterion. For
fair comparison with DB, we will choose Max to yield the
polynomial space bound of $O[n^2d]$. Next we show that Al-
gorithm-3 has this desired polynomial bound.

**Theorem 1:** Algorithm-3 has a space bound of $O[n^2d]$ for
value Max = n(n-1)d.

**Proof:**
- Suppose a new nogood $\neg (\alpha_1 \land \alpha_L)$ is added to $\Gamma$ causing
  it to be full.
- Zero or more nogoods will be deleted from $\Gamma$ of the form
  $\neg (\alpha_1 \land \alpha_L)$, for $\alpha_L \in \alpha_H$.
- If at least one nogood has been deleted then the cache is
  no longer full (and the bound is not exceeded).
- Otherwise, zero nogoods have been deleted. This could
  happen at most once for each possible culprit $\alpha_L$ which
  is $n$ variables times the average size of the domains $d$.
- Therefore if we define Max = $n^2d - nd = n(n-1)d$ then the
  actual size $|\Gamma|$ can never exceed $n^2d$ nogoods at any par-
  ticular time.

**Discussion**

The nogood caching problem identified here manifests
when trying to preserve agent autonomy in AB. [Ba-
yardo & Miranker 96] suggest that in sequential applications
of IB that retaining nogoods which differ from the current
point by no more than a single binding (1-relevance learn-
ing) exhibits worst-case execution behaviour that is within
a constant of unrestricted retention of all nogoods for a partic-
ular variable (dependency directed backtracking). We sus-
pect that this may be true for only the worst case in seque-
tial algorithms but is not true in the distributed case of
AB. The intuition is that sequential depth-first algorithms
tend to focus on exhausting the live domain of a single cul-
prit variable before backtracking and hence changing the de-
fining set of that variable. Thus the defining set changes
bindings slowly and the AA-rule does not affect perfor-
mance too adversely. This is clearly not the case for distrib-
uted algorithms whose agents are allowed to asynchronously change the bindings of their variables. In
the distributed case, defining sets change spontaneously and
unpredictably.

**Conclusion**

Multiagent approaches to solving CSPs are becoming more
prevalent. Constraint solving techniques developed for se-
quential algorithms need to be evaluated carefully before be-
ing applied in the DCSP framework. In this paper, we have
examined the nogood caching scheme developed for DB and
found it unsuitable for maintaining the nogood store in AB.
We showed that this caching scheme discards and forces the
recomputation of nogoods for each agent repeatedly. Two
new nogood caching schemes were suggested which allevi-
ate this problem. Neither scheme preemptively nor unneces-
sarily empties the cache. Two new cache algorithms were
given. The first algorithm has a non-polynomial space
bound while the second algorithm exhibits the same desir-
able polynomial space behaviour as DB. These results are
work in progress and empirical evaluation of these caching
algorithms is underway.

**Acknowledgments**

This work was begun while visiting the Knowledge Systems
Group at Sydney University. I would like to thank Dr. Nor-
man Foo for his gracious hospitality and Dr. Aditya Ghose
for his insightful comments.

**References**

Satisfaction Problems: Experimental and Theoretical
Results. Ph.D. thesis, Dept. of Computer and Information
Systems, Univ. of Oregon.

Bayardo, R.J. & Miranker, D.P. 1996. A Complexity Analy-
sis of Space-Bounded Learning Algorithms for the Con-
straint Satisfaction Problem. In proc. AAAI-96: 13th
National Conf. on Artificial Intelligence, Portland, Oregon,
298-304.


A.I. Research 1, Morgan-Kaufmann, 25-46.

Ginsberg, M. L. & McAllester, D. 1994. GSAT and
Dynamic Backtracking. In proc. 2nd Workshop on Princi-
ples and Practice of Constraint Programming, Orcas Island,
WA.

Distributed Constraint Satisfaction for Formalizing Distrib-
uted Problem Solving, In proc. 12th IEEE Int. Conf. of Dis-

Yokoo, M. 1993. Dynamic Variable/Value Ordering Heuris-
tics for Solving Large-Scale Distributed Constraint Satis-
faction Problems. In proc. 12th Int. Workshop on Dis-
tributed Artificial Intelligence.