Generalizing Inconsistency Learning for Constraint Satisfaction*

Eugene C. Freuder and Richard J. Wallace
Department of Computer Science
University of New Hampshire
Kingsbury Hall M208, College Road
Durham, NH 03824 USA
e-mail: ecf,rjw@cs.unh.edu

Abstract

Constraint satisfaction problems, where values are sought for problem variables subject to restrictions on which combinations of values are acceptable, have many applications in artificial intelligence. Conventional learning methods acquire individual tuples of inconsistent values. These learning experiences can be generalized. We propose a model of generalized learning, based on inconsistency preserving mappings, which is sufficiently focused so as to be computationally cost effective. Rather than recording an individual inconsistency that led to a failure, and looking for that specific inconsistency to recur, we observe the context of a failure, and then look for a related context in which to apply our experience opportunistically. As a result we leverage our learning power. This model is implemented, extended and evaluated using two simple but important classes of constraint problems.

1 Introduction

1.1 Overview

Constraint satisfaction problems (*CSPs*) involve finding values for problem variables subject to constraints on which combinations of values are allowed. They have many applications in artificial intelligence, ranging from design to diagnosis, language understanding to machine vision. A *solution* is an assignment of a value to each variable such that all the constraints are simultaneously satisfied. A *constraint* on a subset of k variables can be regarded as a set of k-tuples of values, where each k-tuple represents an inconsistent choice of values for those variables.

In addition to the inconsistencies in the constraints that are given to define the problem, there are implicit inconsistencies, that become apparent during the search for a solution. In general, a k-tuple of values is *inconsistent*, or a *nogood*[1], if it does not appear in any solution. Solution methods are commonly based on backtrack search. When a failure is encountered during search, there are "learning" methods that can identify and record an individual nogood "responsible" for the failure. While the inference methods employed to extract the nogoods may be relatively sophisticated, the actual learning tends to be rather primitive. There is little if any of the generalized learning from examples or background knowledge that has been employed so successfully in the machine learning community. (There are other interesting applications of learning methods to constraint satisfaction, to synthesize heuristics or algorithms, but we focus here on learning additional constraint information.)

As a simple, motivating example of generalized learning consider five vertices, A, B, C, D and E in a coloring problem with a great many vertices and three colors: red, blue and green. Suppose that there are no edges among these five, but that each of them is connected to vertex F. A person trying to color the vertices in lexical order would observe that while there are no edges between vertices A, B and C, nevertheless there is an implied constraint that if A is red and B is blue, then C cannot be green. A CSP search algorithm could make the same observation. A standard CSP learning algorithm could learn and remember that the triple of values (red, blue, green) for the triple of variables (A, B, C) is inconsistent. A person, however, would be likely to generalize that learning:

- A person could recognize that D and E cannot be green either, when A is red and B blue.
- When encountering a similar configuration among other variables elsewhere in the problem, the person could recognize that this situation had been encountered before.
- A person would understand that the specific colors involved are not critical, they could be black, orange and brown.
- A person might even observe that a similar situation exists when four colors are available (or n colors).

In this paper we introduce a basic form of generalized learning. The principle behind it is this: when a subproblem "learning environment" can be mapped onto another subproblem encountered later in the search with

*This material is based on work supported by the National Science Foundation under Grant No. IRI-9207633

[1]We only consider global nogoods, whose inconsistency does not depend on a context of other choices.

an "inconsistency preserving mapping", then the original learned nogood generalizes to the new situation.

Of course, finding such mappings could easily become more computationally expensive than solving the original problem. To apply the general insight we need to identify learning contexts that are general enough to be interesting, but restricted enough to make generalization cost effective. We identify a basic context that meets these criteria (which encompasses, in particular, the first two examples of generalization listed above), and then begin to relax the restrictions.

We validate the utility of generalization in two specific contexts. We start by considering "coloring" problems in which the single permitted constraint is inequality. These "constraints of difference" [Regin, 1994] are of considerable interest, and in fact model basic coloring, scheduling and resource allocation problems. Next we add the ordering constraints: $<, >, \leq, \geq, =$.

1.2 Relation to Previous Work

There is a stream of work on CSP nogood learning (see [Frost and Dechter, 1994; Schiex and Verfaillie, 1993] for recent examples) with connections to truth maintenance (e.g. [Smith and Kelleher, 1988]). There is a stream of work on CSP symmetry (see [Ellman, 1993; Puget, 1993] for recent examples), which extends at least as far back as ([Fillmore and Williamson, 1974]). One of the current authors did some tentative work earlier on subgraph isomorphism in constraint graphs [Freuder, 1984]. Benhamou [Benhamou, 1994] uses symmetrical values and search branches to improve CSP algorithms.

The current work brings these two streams together, using isomorphic mappings to generalize learned nogoods. We find local mappings, dynamically during search, between subproblems, in which some of the variables have been instantiated to specific value choices. The pruning done by the basic form of generalization that we implement would be subsumed by some forms of look-ahead (though not by forward checking alone) [Haralick and Elliott, 1980]. However, generalization accomplishes the pruning in a more targeted manner; our experimental results show that generalization can be more cost effective than the additional look-ahead.

2 Principle

Generalization will be based on inconsistency preserving mappings. A *mapping* takes a *value/variable pair*, e.g. red/A, value red for variable A, into another value/variable pair. An *inconsistency preserving mapping*, f, from a *source subproblem*, S, to a *destination subproblem*, D, is a function from the value/variable pairs of S into the value/variable pairs of D, such that if any v/V is inconsistent with any u/U in S, then f(v/V) will be inconsistent with f(u/U) in D. Clearly, inconsistency preserving mappings can permit us to generalize the application of learned nogoods by mapping a nogood acquired in one subproblem into a situation encountered in another subproblem.

Just as clearly, looking for these mappings could easily be computationally counterproductive. We can focus our attention along a number of axes: semantic, syntactic and pragmatic. We choose a very restricted starting point on each of these axes, to establish a basic model of generalization learning, and then begin to move outward. We begin with with *value-identity mappings* such that a value/variable pair, v/V, can only map into a pair, v/U, involving the same value. We assume that in a given problem all variable domains are the same, and all the given constraints are the same, and binary (involves only two variables). We use the fundamental "deadend learning" context to establish the structure of the source subproblem, and the basic backtrack search structure to propose destination subproblems.

Instances of the prototypical source and destination problems, with an inconsistency preserving mapping between them, are shown in Figure 1. The source subproblem consists of a hub variable and k spoke variables. Each spoke variable domain has been reduced to a single value; all these values are deemed mutually consistent by the given problem constraints. Every value in the hub variable domain is inconsistent with at least one of the spoke variable values. This is the basic CSP deadend learning situation, in which we record that the spoke variable values (or some subset of them) form a learned nogood.[2] Here, for simplicity, we assume that the hub variable in the destination problem corresponds to the current variable in the search, the one for which we are about to choose an instantiation. The values from k-1 of the spoke variables in the source map into k-1 previously chosen values in the destination. The kth source spoke variable corresponds to a "future" uninstantiated variable at the destination. Applying the discovered generalization is done by pruning the value in the source subproblem corresponding to the value at the kth spoke variable. (Note that there may be additional "past", instantiated variables, or future, uninstantiated ones that constrain the current variable. The other future variables offer additional opportunities for generalization pruning.)

Given the assumptions that we have made here, it is easy to see that this is an inconsistency preserving mapping, and that to discover it all we have to do is observe that k-1 of the source spoke values appear in instantiated variables that share a constraint with the current variable in the destination subproblem. (For example, there may be constraints between spoke variables in the source, but they are irrelevant now since the spoke variable values are all mutually consistent.) Discovering source problems requires essentially the same effort as deadend learning; the effort expended in looking for an opportunity to generalize is strictly circumscribed.

In the next sections we will implement and evaluate this basic generalization mechanism. We start by considering "coloring" problems in which the single permitted constraint is inequality. These problems precisely fit the basic model we have just introduced. Next we add the

[2] In more advanced learning the deadend can consist of a subproblem rather than a single variable. This is beyond the scope of the present paper; however, we believe we can generalize such advanced learning, and avoid combinatorial explosion by limiting the size of the failed subproblem.

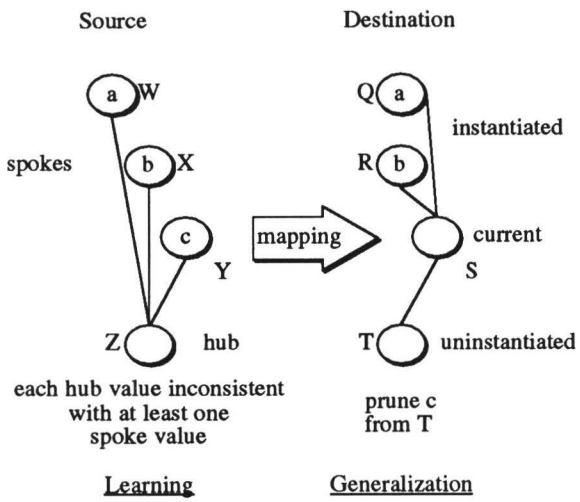

Figure 1: Generalization.

ordering constraints. This motivates us to relax the restrictions of the basic model. We observe that mappings will be inconsistency preserving as before as long as the constraints in the destination are more restrictive than the corresponding constraints in the source (e.g. "<" is more restrictive than "≤"). We also move beyond value-identity mapping in this context.

3 Nogood Learning and Generalization

Generalization can, of course, be used with a variety of CSP algorithms. In the present paper, a forward checking engine is used to evaluate the effect of adding generalization. As part of the overall comparison, nogood learning without generalization [Dechter, 1990] [Schiex and Verfaillie, 1993] is also considered.

We first describe our nogood learning procedure, and then discuss the ways in which learning based on generalized nogoods differs from it. When nogood learning is used with forward checking, preclusion based on k-ary nogoods is performed in tandem with ordinary preclusion (Figure 2). (In "ordinary preclusion", values in future (uninstantiated) domains that are inconsistent with the latest value chosen for instantiation are discarded, or precluded.) For fixed order search, k-ary nogoods are ordered by their variables in accordance with the search order. Each nogood is associated with the penultimate variable in its ordered set. At this point in search, a match with previous assignments, plus the value currently being considered for the current variable, allows preclusion of the domain of the last variable in the nogood.

Two variants of nogood learning will be considered. One is based on the set of variables that are adjacent to one that has been wiped out. This is the procedure that [Frost and Dechter, 1994] call graph-based nogood learning. The other is based on the *preclusion set*, i.e., the variables whose assignments led to deletions of values in the domain that has been wiped out. This is called the jump-back set by [Frost and Dechter, 1994] and the killer set by [Schiex and Verfaillie, 1993]. In neither case is the nogood set minimalized. (But for coloring problems the preclusion set is a minimal nogood.) Nogoods associated with one variable are checked in order of the most recently discovered. (A few tests were made with the reverse order, with similar results.)

Search:
Choose an uninstantiated variable v_i
  for each value a in domain d_i
    if there is a nogood associated with v_i
    perform k-ary preclusion
    if no wipeout, perform ordinary preclusion
      if wipeout occurs
        set up nogood associated with next
        to last variable in preclusion set

K-ary Preclusion:
for each nogood associated with v_i
  test whether current assignments to nogood variables
  1 to $k-1$ equal corresponding nogood values
    if nogood matches current assignments
      delete nogood value from domain of kth
      variable in nogood

Figure 2: Scheme for Nogood Learning.

In the algorithms discussed here nogood building is based solely on ordinary preclusion. Thus, results of preclusion due to k-ary nogoods are not used. This would have required further elaborations to search for the relevant variables and to combine them properly. Since the nogoods will be larger than ones based on ordinary preclusion, they are probably not as useful. This is because the probability that a nogood matches a current partial assignment decreases as the size of the nogood increases. Some overhead is incurred because the program has to check that domain wipeout did not involve k-ary nogoods.

The procedure for building generalized nogoods is similar to that used for ordinary nogood learning (Figure 3). Nogoods are built following wipeout of a domain. But in this case the nogood consists of a set of disallowed values (one per domain) together with the conditions under which they are disallowed. In the general case these conditions are: the type of constraint and the original set of values in the domain that was wiped out. If a particular condition is consistent for the entire problem, then it does not need to be specified. For example, if the original domains of the problem are identical, then a nogood will hold for any subproblem having the necessary pattern of constraints. If both the domains and the constraints are constant for a problem, then only the values need to be specified.

For coloring problems, inconsistency mapping was done so that the hub variables were the uninstantiated (i.e., future) variables adjacent to the variable currently considered for instantiation. In other words, generalization was based on a look-ahead by one procedure. In coloring problems, if the current variable itself is used as

the hub, then preclusion based on nogoods is completely redundant with ordinary preclusion. This is because nogood preclusion will apply only in those cases where the domain of the current variable has been reduced to the one value that can be precluded from the adjacent domains. But, then, the value in the current domain will also preclude the same values from the adjacent domains, with the same (total) number of constraint checks. Since look-ahead by one proved to be a successful strategy, it was used with all sets of problems.

```
Search:
Choose an uninstantiated variable v_i
  for each value a in domain d_i
    if there is a nogood associated with v_i
    perform 'adjacent' k-ary preclusion
    if no wipeout, perform ordinary preclusion
      if wipeout occurs
        set up nogood associated with next
        to last variable in preclusion set

Adjacent K-ary Preclusion:
For each variable v_j (of sufficient degree) adjacent to v_i
  perform generalized k-ary preclusion

Generalized K-ary Preclusion:
for each generalized nogood
  test current assignments of instantiated variables
  adjacent to v_j to determine applicability
    if k - 1 nogood values can be matched with current
    assignments for each uninstantiated variable
    adjacent to v_j
      delete kth nogood value from domain of
      that variable
```

Figure 3: Scheme for Generalized Nogood Learning (with look-ahead by one).

As with ordinary nogood learning, variants based on adjacency sets and on preclusion sets were included in the tests. In connection with adjacency sets, two further cases are considered. In the first, there is no attempt to minimalize the set of nogoods and all are used in attempting to preclude values. The second case involved minimalization. For nogood learning this is probably too expensive [Dechter, 1990; Frost and Dechter, 1994]. But, if there is only one set of generalized nogoods, as in the cases considered here, minimalization may be cost-effective.

4 Experimental Tests

4.1 Problems Tested

Generalization in k-coloring problems is carried out as described in the previous section. If a nogood is discovered during search, it can be used with any variable having k or more adjacent variables, to delete values from some of the adjacent nodes. Of course, some procedures (those based on adjacency sets) may discover nogoods that are multisets of the k colors, and these can be generalized in the same fashion.

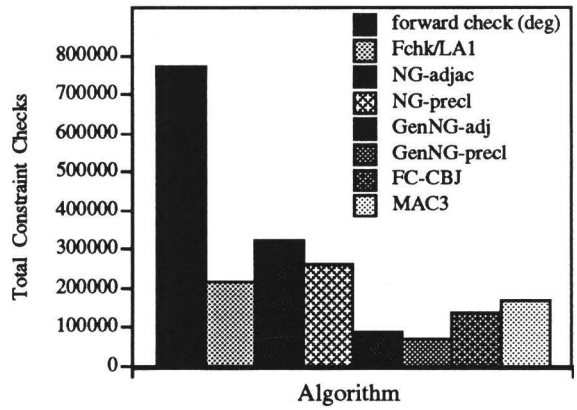

Figure 4: Performance on 3-color problems, density = 0.03. Means for 50 problems. Legend maps onto columns of graph when former is rotated counterclockwise.

For problems with ordering constraints based on relational operators ("relop" problems), nogoods are discovered in the same fashion as with coloring problems. For ordinary nogood learning, use of nogoods during search is also identical. However, in order to generalize properly, information about constraints must be considered in addition to nogood values. In the general case these nogoods are applied according to the mapping principles described in Section 2. This means that each constraint associated with a nogood value must match (by subsumption) a constraint in the subproblem being considered, and that the nogood value must not be more constraining than the current instantiation.

For example, suppose that the domains of a problem are all {1, 2, 3, 4} and that the assignment for variable v_i must be greater or equal to the value of variable v_k, and the assignment for variable v_j must be equal to the value of v_k. In this case, assigning values 3 and 4, respectively, to v_i and v_j results in a wipeout of the domain of v_k. This nogood can be represented by the tuple, (($\geq$ 3) (= 4)), whose elements can be thought of as partial Lisp expressions with the second argument unspecified. In addition to matching any subproblem with the same constraints and values, the first element of this nogood will match an assignment of 2 to a variable associated with a $\geq$ constraint, an assignment of 3 to a variable with a $>$ constraint, and so forth.

Clearly, the effort to find a general match in this case may be too costly. In particular, finding an adequate mapping may require backtracking when a nogood element can be matched to more than one element in the subproblem. However, the comparison criteria can be tightened, and search effort may still be reduced with acceptable overhead. For example, the comparison can be discontinued if a nogood element does not match any of the remaining instantiations, to avoid backtracking. In addition, matches can be restricted to identity mapping of constraints (here, exact matches for the relational

operators). And, in addition to these matching strategies, targeting strategies can be used to limit the number of contexts that are checked for mapping. In sum, what generalization offers in this context is a set of opportunities for nogood matching with attendant tradeoffs between search reduction and complexity of the additional computation.

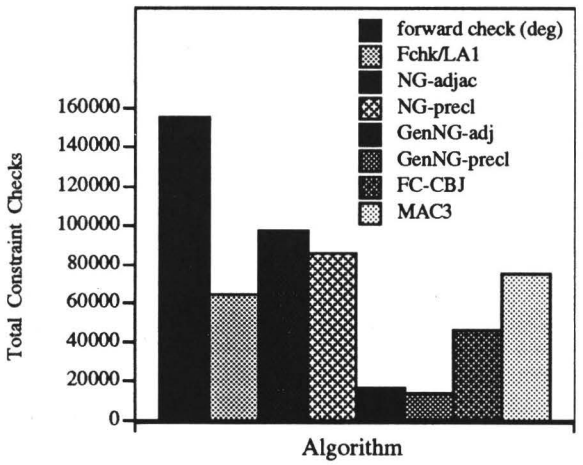

Figure 5: Performance on 3-color problems, density = 0.04. Means for 50 problems. Same relation between legend and figure as in Figure 4.

4.2 Experimental Methods

Coloring problems had either three or four colors. Three-color problems had 100 variables and an expected graph density of 0.02, 0.03 or 0.04; Four-color problems had 70 variables and an expected density of 0.09. (Here, density is measured in terms of edges added to a spanning tree.) The 3-color densities span the critical region for problem complexity. In accordance with this, problem difficulty increases from 0.02 to 0.03 density, and decreases again at 0.04 density. The percentage of problems with solutions at successively higher densities was 98, 12 and 0. Four-color problems were also in the critical region; 56 percent of these problems had solutions.

Relop problems had 45 variables and expected densities of 0.01 and 0.02. The percentage of problems with solutions at successively higher densities was 46 and 4.

Both types of problem were generated according to a random model for arc inclusion. Construction began with a spanning tree in which successive nodes were chosen at random and connected at random to nodes already in the tree. Then each remaining edge was chosen with a probability equal to the expected density. For relop problems, the type of each constraint was chosen at random from among the six possibilities. Fifty problems were generated for each sample.

The basic experimental comparison was with forward checking. In these experiments, each algorithm was run with a fixed variable ordering based on (maximum) degree of a node in the constraint graph for the CSP. (We

Table 1: Timing Results for 3-Color Problems (Sec)

	graph density		
	.02	.03	.04
forward check	15.5	23.8	4.7
FC+LA-1	(11.4)	(30.8)	(9.0)
NG-adjac/all	2.8	8.8	2.7
NG-precl/all	2.5	7.9	2.7
GenNG-adj/all	5.9	5.3	1.2
GenNG-adj/min	5.7	4.3	1.1
GenNG-precl	5.9	4.2	1.0
FC-CBJ	0.4	5.8	2.0
MAC3	5.0	9.3	4.4

Note. Entries are sample means. FC+LA-1 times are for a slower implementation than the others.

are currently implementing dynamic ordering by minimal domain size; this should still allow improvement due to nogood learning, as [Frost and Dechter, 1994] have shown.) Forward checking with look-ahead (FC+LA-1) was also tested to determine the effect of this procedure, which was used with generalized nogood testing. Look-ahead was done using distance bounded relaxation [Freuder and Wallace, 1991] with distance = 1, which was the same as the distance used with the nogood algorithms. Results were also compared with two of the most powerful techniques for solving CSPs: forward checking with conflict-based backjumping (FC-CBJ) [Prosser, 1993] and maintained arc consistency (MAC3) [Sabin and Freuder, 1994]. In the latter case, an AC-3-like procedure was used to maintain arc consistency, in order to have a measure (consistency checks) that allowed comparison with the other algorithms.

For coloring problems, both ordinary and generalized nogood learning (NG and GenNG or Gen in the tables) were tested using either adjacency sets or preclusion sets ("adj" and "precl" in tables); adjacency sets were either full or minimalized when used with generalized nogoods. For relop problems, testing was done with preclusion sets only. The following types of generalization were also tested: (i) simple identity matching (GenNG-id), (ii) identity matching of constraints (relational operators) and set/subset matching of values (GenNG-idop), (iii) set/subset matching of both operators and values. For the last two categories, a kind of minimalization was also carried out, according to the following procedure. If k-1 elements in two nogoods were identical, the kth elements were matched for operators. If both operators were in the set $\{>, \geq\}$ or in $\{<, \leq\}$, then the element whose value was more inclusive was retained.

Three measures of performance were used: backtracks, constraint checks and total search time. The number of backtracks, i.e., the number of times the values in a domain were exhausted causing search to back up, is an indication of the size of the search tree. Number of consistency (or constraint) checks often gives a better overall view of performance. All algorithms did some consistency checking as part of the preclusion done by forward checking. Algorithms that incorporated nogood learning

also checked k-ary nogoods. Algorithms that employed look-ahead also did consistency testing between values in domains of future, uninstantiated variables. Naturally, k-ary constraint checks differ from ordinary constraint checks in the time required, and the time required to test ordinary nogoods is different from the time to test generalized nogoods. For these problems simple nogood testing was faster than ordinary constraint checking, partly because the nogoods were small and look-up operations were simpler. (Rapidity of failure when testing successive nogood variables may also have been a factor.) Generalized nogood testing was more involved, since the set of variables and their order were not known in advance; hence, this operation was slower than ordinary constraint checking. (These differences are most evident in the data of Table 2.)

Experiments were run on a DEC Alpha (DEC 3000 M300LX). Algorithms were coded in Common Lisp, using Lispworks by Harlequin. All solutions were tested for validity; in addition, solutions produced by the new algorithms described in this paper were compared with those produced by a version of forward checking that has been thoroughly tested in previous work. (All of these algorithms should find the same first solution to a problem if the same variable and value orderings are used.)

Table 2: Results for 4-Color Problems (Density = .09)

	bts	constraint checks			time
		fc	ng	total	
forward chk	470	8,505		8,505	231
NG-adj/all	151	2,579	29,318	31,897	311
NG-prec/all	103	1,820	9,782	11,602	158
Gen-adj/all	39	711	2,147	2,858	221
Gen-adj/min	39	715	411	1,127	72
Gen-prec	39	715	411	1,127	76
FC-CBJ	57	2,725		2,725	105
MAC3	1	91	3,707	3,798	150

Note. Means for 50 problems. Backtracks and constraint checks in thousands. Times in sec. Abbreviations for algorithms in text.

4.3 Results

Coloring Problems

For all problem sets, nogood learning improved on forward checking alone (Figures 4-5, Tables 1-2). In most cases generalized nogood learning was better than ordinary nogood learning; the only exception was for 3-color problems with expected density = 0.02 (see Table 1). Comparisons involving look-ahead show that this technique may have been partly responsible for the improvement found with generalized nogood learning, but the latter was superior to straight look-ahead in situations in which it was also better than ordinary nogood learning. Generalized nogood learning also outperformed forward checking with conflict-directed backjumping or maintained arc consistency on most problem sets, including those in the critical complexity regions.

As would be expected, minimalizing nogoods reduced the number of constraint checks. Timing data also indicate that overall performance was improved, so that minimalization used with generalized nogood learning was cost-effective in this case. Using preclusion sets was more effective than using adjacency sets for both ordinary and generalized nogood learning, and this difference was greater for the harder 4-color problems. But for generalized nogood learning, minimalization made the two almost identical with respect to search time and measured operations.

Relop Problems

For these problems, nogood learning was again superior to forward checking alone (Figure 6). The effect of using generalized nogoods depended on the character of the match and the use of minimalization. With pure identity matching, performance was actually worse, showing that a simple matching strategy corresponding to the one used with coloring problems was not sufficient in this case. More sophisticated matching based on set/subset relations was much more successful, and performance was further improved by the minimalization procedure. (Examination of data structures during runs confirmed that this procedure did reduce the proliferation of nogoods found during search.) Although the results were not as good as those for ordinary nogood learning, they show that generalization can improve on the basic forward checking engine even for problems with a fairly complicated pattern of constraints.

The targeting strategies mentioned at the end of Section 4.1 have not yet been implemented and these may improve performance further. In particular, a simple 'match and store' strategy, in which matching nogoods are stored in a manner similar to ordinary nogoods, may reduce the amount of redundant matching during search.

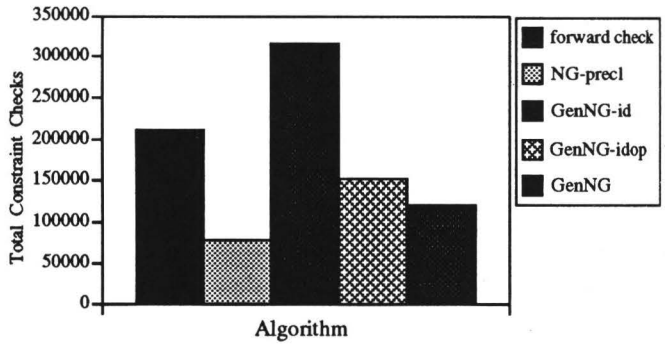

Figure 6: Performance on relop problems, density = 0.02. Means for 50 problems.

5 Conclusion

By using inference, background knowledge, and recognizing structural similarities within and among problems, we can potentially leverage our learning power. We have identified a basic principle that supports generalized learning, and begun to identify specific contexts in which the principle can be profitably applied.

References

[Benhamou, 1994] B. Benhamou. Study of symmetry in constraint satisfaction problems. In *Second Workshop on Principles and Practice of Constraint Programming, PPCP-94*, pages 246–254, 1994.

[Dechter, 1990] R. Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[Ellman, 1993] T. Ellman. Abstraction via approximate symmetry. In *Proceedings IJCAI-93*, pages 916–921, 1993.

[Fillmore and Williamson, 1974] J. P. Fillmore and S. G. Williamson. On backtracking: A combinatorial description of the algorithm. *SIAM Journal of Computing*, 3:41–55, 1974.

[Freuder and Wallace, 1991] E. C. Freuder and R. J. Wallace. Selective relaxation for constraint satisfaction problems. In *Int'l Conference on Tools for Artificial Intelligence, TAI-91*, pages 331–339, 1991.

[Freuder, 1984] E. C. Freuder. *Utilizing subgraph isomorphism in constraint graphs.* Tech. Rept. 84-13, Computer Science Dept, Univ New Hampshire, 1984.

[Frost and Dechter, 1994] D. Frost and R. Dechter. Dead-end driven learning. In *Proceedings AAAI-94*, pages 294–300, 1994.

[Haralick and Elliott, 1980] R. Haralick and G. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980.

[Prosser, 1993] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9:268–299, 1993.

[Puget, 1993] J.-F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Seventh International Symposium, ISMIS-93*, 1993.

[Regin, 1994] J.-C. Regin. A filtering algorithm for constraints of difference in csps. In *Proceedings AAAI-94*, pages 362–367, 1994.

[Sabin and Freuder, 1994] D. Sabin and E. C. Freuder. Contradicting conventional wisdom in constraint satisfaction. In *Proceedings ECAI-94*, pages 125–129, 1994.

[Schiex and Verfaillie, 1993] T. Schiex and G. Verfaillie. Nogood recording for static and dynamic constraint satisfaction problems. In *Int'l Conference on Tools with Artificial Intelligence, TAI-93*, pages 48–55, 1993.

[Smith and Kelleher, 1988] B. Smith and G. Kelleher. *Reason Maintenance Systems and Their Applications.* Ellis Horwood, Chichester, England, 1988.