Generalized NoGoods in CSPs

George Katsirelos
University of Toronto
Toronto, Canada
gkatsi@cs.toronto.edu

Fahiem Bacchus
University of Toronto
Toronto, Canada
fbacchus@cs.toronto.edu

Abstract

Although nogood learning in CSPs and clause learning in SAT are formally equivalent, nogood learning has not been as successful a technique in CSP solvers as clause learning has been for SAT solvers. We show that part of the reason for this discrepancy is that nogoods in CSPs (as standardly defined) are too restrictive. In this paper we demonstrate that these restrictions can be lifted so that a CSP solver can learn more general and powerful nogoods. Nogoods generalized in this manner yield a provably more powerful CSP solver. We also demonstrate how generalized nogoods facilitate learning useful nogoods from global constraints. Finally, we demonstrate empirically that generalized nogoods can yield significant improvements in performance.

Introduction

Backtracking search is one of the main algorithms for solving constraint satisfaction problems (CSPs), and nogood (or conflict) learning is a standard technique for improving backtracking search (Dechter 1990). The technique of learning nogoods was imported into SAT solvers by (Bayardo & Schrag 1997). In the SAT context nogoods correspond to clauses, and better ways of learning and exploiting large numbers of clauses during search (Moskewicz *et al.* 2001) have proved to be revolutionary to SAT solving technology.

In contrast, nogoods have not had as large an impact on CSP solvers. For example, most commercial CSP solvers do not use nogood learning. Part of the reason for this is that these solvers make heavy use of special purpose algorithms (propagators) for achieving generalized arc consistency (GAC) on constraints of large arity. To date it has been difficult to learn useful nogoods from such propagators. As a result nogoods have not yielded sufficient performance improvements on practical problems to justify their overhead.

In this paper we show that part of the reason for these lackluster performance improvements arises from the manner in which nogoods have standardly been defined. In particular, we show that the standard definition of nogoods is unnecessarily restricted, and that this restriction can be lifted. The resulting generalized notion of a nogood *provably* increases the power of backtrack search. Furthermore, we show how generalized nogood allow us to learn more useful nogoods from global constraints, thus helping to address the main stumbling block for employing nogoods in modern "propagator-heavy" CSP solvers. Finally, we verify empirically that generalized nogoods can in practice often yield a significant improvements (orders of magnitude) in CSP solving times.

Background

A CSP consists of a set of variables $\{V_1, \ldots, V_n\}$, a domain of values for each variable $Dom[V_i]$, and a set of constraints $\{C_1, \ldots, C_m\}$. A variable V can be assigned any value a from its domain, denoted by $V \leftarrow a$. An **assignment set** is a set of assignments $\mathcal{A} = \{X_1 \leftarrow a_1, \ldots, X_k \leftarrow a_k\}$ such that no variable is assigned more than one value. We use $VarsOf(\mathcal{A})$ to denote the set of variables assigned values in $\mathcal{A}$. Each constraint C_i consists of a set of variables called its **scope**, $scope(C_i)$, and a set of assignment sets. Each of these assignments sets specifies an assignment to the variables of $scope(C)$ that satisfies C. We say that an assignment set $\mathcal{A}$ is **consistent** if it satisfies all constraints it covers: $\forall C. scope(C) \subseteq VarsOf(\mathcal{A}) \Rightarrow \exists \mathcal{A}'. \mathcal{A}' \in C \wedge \mathcal{A}' \subseteq \mathcal{A}$. A **solution** to the CSP is a consistent assignment set containing all of the variables of the CSP.

The standard definition of a **nogood** or conflict set (Dechter 1990) is an assignment set that is not contained in any solution: a nogood cannot be extended to a solution. For example, any assignment set that violates a constraint is a nogood. Similarly, a CSP has no solution if and only if the empty set of assignments is a nogood.

Nogoods can be learned during search, used for non-chronological backtracking, and also stored and used to prune future parts of the search tree. To make this precise we consider nogood learning within the forward checking algorithm (FCCBJ+S) shown in Table 1. This algorithm uses nogoods to perform conflict directed backjumping (FCCBJ) (Prosser 1993) as well as storing nogoods for future use (+S to denote with storing of standard nogoods).

As in ordinary FC, `forwardCheck` detects and prunes all values of the unassigned variables that in conjunction with the current prefix of assignments violate a constraint. We also record, in the set $NG[d, V]$, the nogood that caused the value $d \in Dom[V]$ to be pruned. That is, if $V \leftarrow d$ is pruned because it violates a constraint C, then we set $NG[d, v] = \{V \leftarrow d, X_1 \leftarrow x_1, ..., X_k \leftarrow x_k\}$, where $scope(C) = \{V, X_1, \ldots, X_k\}$ and x_i is the currently assigned value of X_i. That is, $NG[d, V]$ is set to be the assignment set that violates C.

```
int FCCBJ+S(L)
1.  If all variables assigned, report solution and exit.
2.  choose next variable V.
3.  If V has had all of its values pruned, choose V.
4.  foreach d∈Dom(V) s.t. d is not pruned
5.     V← d
6.     forwardCheck() and NoGoodProp()
7.     if ((btL = FCCBJ+S(L+1)) < L)
8.        undo(V← d) and return(btL)
9.     else undo(V← d)
10. NewNG = ⋃_{d∈Dom(V)}(NG[d,V] − {V← d})
11. storeNoGood(NewNg)
12. X← a = deepest assignment in NewNG
13. NG[X,a] = NewNG
14. btL = level X← a was made.
15. return(btL)
```

Table 1: Forward checking with conflict directed backjumping and standard nogoods.

The additional function NoGoodProp is used to unit propagate the previously stored nogoods. In particular, if $N = \{V{\leftarrow}d, X_1{\leftarrow}x_1, \ldots, X_k{\leftarrow}x_k\}$ is among the stored nogoods and the assignments $X_1{\leftarrow}x_1, \ldots X_k{\leftarrow}x_k$ have all been made in the current prefix, then NoGoodProp will use this nogood to prune $V{\leftarrow}d$ and will set $NG[V,d] = N$.

After propagation, FCCBJ+S is recursively invoked at the next level. If propagation resulted in a domain wipeout (a variable having all of its values pruned), that variable will necessarily be chosen in the next recursive invocation (line 3).[1]. If the recursive invocation indicates backtracking to a higher level (via its returned value btL), FCCBJ+S continues to return until it reaches that level.

Finally, if all values of V are tried (line 10), we have for every value $d \in Dom[V]$ that either (a) d was pruned by forwardCheck at a higher level, in which case $NG[d,V]$ has been set, or (b) $V{\leftarrow}d$ was attempted and the recursive call to FCCBJ+S (line 7) failed back to this level, in which case $NG[d,V]$ has been set at a deeper level (line 13). So at line 10 all values of V have $NG[d,V]$ set.

All of these nogoods (minus their assignments to V) can then be combined (line 10). The result must be a new nogood since if any solution S extends *NewNG*, S must necessarily extend one of $NG[d,V]$ since S must include some assignment to V. We can then store this new nogood for use along future paths, and use it to perform non-chronological backtracking. In particular, we can backtrack to undo its deepest assignment $X{\leftarrow}a$ and set the reason that $X{\leftarrow}a$ failed as being this new nogood (lines 12-15).

Generalized Nogoods

Consider the constraint $X + Y < Z$ with $Dom[X] = Dom[Y] = \{0,1,2\}$, and $Dom[Z] = \{1,2,3\}$. This constraint is violated by the assignment set $A = \{X{\leftarrow}0, Y{\leftarrow}1, Z{\leftarrow}1\}$, thus A is a nogood. It is also impossible to satisfy C if we assign $Z{\leftarrow}1$ and at the same time prune 0 from the domain of X. However, this condition cannot be expressed as a standard nogood, as we haven't actually assigned a value to X.

We can generalize nogoods so that they can express such conditions by allowing nogoods to contain either assignments or non-assignments. Non-assignments correspond to pruned values—once the a is pruned from the domain of V we can no longer make the assignment $V{\leftarrow}a$. With generalized nogoods we can then express the above condition with the set $\{X{\not\leftarrow}0, Z{\leftarrow}1\}$. It turns out that although syntactically distinct, this generalization is equivalent to the one proposed in (Focacci and Milano 2001). However, the authors of that paper investigated generalized nogoods only in the context of symmetry breaking. The results we obtain in this paper are disjoint from their previous results.

In the next section we prove that generalized nogoods increases the power of backtracking search over standard nogoods. For now however, we note two things about generalized nogoods. First they can compactly represent a possibly exponentially sized set of standard nogoods. For example, the above generalized nogood compactly captures the nogoods $\{X{\leftarrow}1, Z{\leftarrow}1\}$ and $\{X{\leftarrow}2, Z{\leftarrow}1\}$. When the generalized nogood contains multiple non-assignments the set of standard nogoods it covers grows exponentially. For example, if all of the variables $V_1, \ldots, V_{10}$ have domain $\{0,1,2,3,4,5\}$, then the generalized nogood $\{X{\leftarrow}1, V_1{\not\leftarrow}0, \ldots V_{10}{\not\leftarrow}0\}$ captures $6^{10} - 1$ different standard nogoods. Second, a generalized nogood can prune paths in the search tree that the set of standard nogoods it captures cannot. For example, $\{X{\leftarrow}1, V_1{\not\leftarrow}0, \ldots V_{10}{\not\leftarrow}0\}$ can prune paths where none of the V_i have yet been assigned, while none of the $10^6 - 1$ standard nogoods it captures can be activated without instantiating all but one of its elements and then performing nogood propagation. We now proceed to a formalization of generalized nogoods.

A solution to a CSP assigns a unique value to every variable, and thus implicitly prunes every other value in the variable's domain. An **expanded solution** to a CSP is a solution to which all true non-assignments (prunings) have been added. For example, if the CSP contains only the variables X and Y each with the domain of values $\{1,2,3\}$, then the solution $\{X{\leftarrow}1, Y{\leftarrow}2\}$ corresponds to the expanded solution $\{X{\leftarrow}1, X{\not\leftarrow}2, X{\not\leftarrow}3, Y{\not\leftarrow}1, Y{\leftarrow}2, Y{\not\leftarrow}3\}$. We can now define a **generalized nogood** to be any set of assignments and/or non-assignments that is not contained in any expanded solution. Standard nogoods are a subset of the generalized nogoods under this definition.

Backtracking search can be adapted so that it learns new generalized nogoods (g-nogoods) during search. To make this precise we first make a few observations. (a) Any extended solution must contain one of $X{\leftarrow}a$ or $X{\not\leftarrow}a$: if the extended solution does not contain $X{\leftarrow}a$ it must contain some other assignment to X and thus must contain $X{\not\leftarrow}a$. (b) If NG_1 and NG_2 are two g-nogoods containing $X{\leftarrow}a \in NG_1$ and $X{\not\leftarrow}a \in NG_2$, then $NG_3 = (NG_1 \cup NG_2) - \{X{\leftarrow}a, X{\not\leftarrow}a\}$ is a new g-nogood: if S is an expanded solution extending NG_3 then since S must contain

[1] If there is more than one wiped out variable correctness is not affected by which one is chosen.

one of $X \leftarrow a$ or $X \not\leftarrow a$ it must also extend one of NG_1 or NG_2 contradicting the fact that these sets are g-nogoods.

Next, when we have a store of g-nogoods, `NoGoodProp` might force assignments as well as non-assignments, and that there might be chains of such forced (non) assignments. For example, if (1) $\{Y \leftarrow a, Y \leftarrow b\}$, (2) $\{X \not\leftarrow a, Y \not\leftarrow b\}$, and (3) $\{Z \leftarrow b, X \leftarrow a\}$ are g-nogoods in the store, and we make the assignment $Y \leftarrow a$, `NoGoodProp` will force the pruning $Y \not\leftarrow b$ ($Y \leftarrow b$ must be false) using nogood 1, then the assignment $X \leftarrow a$ using nogood 2, and then the pruning $Z \not\leftarrow b$ using nogood 3.[2]

We say that the **depth** of the assignment $V \leftarrow d$ that occurs at line 5 of FCCBJ+S is $L.0$ (where L is the input to FCCBJ+S). The other (non)-assignments forced by `forwardCheck` and `NoGoodProp` occur in a sequence after this assignment, and the depth of each of these is $L.j$ where j is the order in which they occur. In the previous example, *depth*$(Y \leftarrow a) = L.0$ (for some L), *depth*$(Y \not\leftarrow b) = L.1$, *depth*$(X \leftarrow a) = L.2$, and *depth*$(Z \not\leftarrow b) = L.3$.

We say that (non)-assignment A with *depth*$(A) = H.i$ is **deeper** than (non)-assignment B with *depth*$(B) = L.j$, if $H > L$ or if $H = L \wedge i > j$. Also all assignments with depth $H.0$ for some H are called **decision assignments**, and all other (non)-assignments are called **forced**. All forced (non)-assignments are labeled by an associated g-nogood containing their complement. For instance, in our previous example for $X \leftarrow a$ we have $NG[a, X] = \{X \not\leftarrow a, Y \not\leftarrow b\}$. The labeling g-nogood for forced (non)-assignments is set by `forwardCheck` or by `NoGoodProp`.

FCCBJ+S can now be modified to learn generalized nogoods. Using $\rightleftharpoons$ to denote $\leftarrow$ or $\not\leftarrow$ we replace line 10 by the new lines

```
10.1 NewNG = {V ≠ d | d ∈ Dom[V]}
10.2 do
10.3    X ⇌ a = deepest (non)-assignment in NewNG
10.4    if X ⇌ a is not a decision assignment
10.5       NewNG = NewNG ∪ (NG[a, X] − {X ⇌ a})
10.6    until X ⇌ a is a decision assignment
```

We call this scheme for learning a new g-nogood a **first-decision scheme**. It iteratively replaces the deepest (non)-assignments the implicit "variable must have a value" nogood until it arrives at a g-nogood containing a decision assignment. This iterative replacement generally leaves a number of the original non-assignments $V \not\leftarrow d$ (line 10.1) in *NewNG*, i.e., the scheme learns a g-nogood rather than an s-nogood. This new g-nogood can then be used just as before to backtrack and to label the assignment we undo on backtrack (lines 11-15). From the observations above it is clear that *NewNG* is a valid g-nogood at each iteration. It is also possible to show that the process must terminate with *NewNG* containing a single decision assignment. However, space precludes us from providing a proof of this latter fact. In sketch however, this scheme for learning new g-nogoods can be shown to be sound, and to converge, by mapping the CSP to a proposition theory. Under that mapping nogoods become clauses, and nogood learning becomes a sequence of resolution steps.

There are many other possible schemes for learning generalized nogoods, and we have implemented a few others. However, all the empirical results provided below were derived using the first-decision learning scheme. Alternate schemes correspond to different strategies for applying resolution, and they are very similar to the different schemes that can be proposed for clause learning (Zhang *et al.* 2001). Interesting, the 1-UIP scheme that is popular in SAT solving did not prove to be as empirically effective as the first-decision scheme.

[2] Such chains of unit propagation are only possible with g-nogoods. Standard nogoods (s-nogoods) only force prunings which can only inactivate other s-nogoods: they can never make another s-nogood unit.

Generalized nogoods are more powerful

As mentioned above g-nogoods can compactly encode large numbers of s-nogoods, and prune paths not pruned by any of these s-nogoods. More interestingly, however, is that it is possible to prove a concrete result about the power g-nogoods add to backtracking search.

First consider the family of CSP backtracking algorithms $\mathcal{F}$ that perform any combination of the standard techniques for constraint propagation (e.g., maintaining arc consistency), intelligent backtracking (e.g., conflict directed backjumping), dynamic variable ordering, and dynamic value ordering. $\mathcal{F}$ includes such standard algorithms as MAC, FC, CBJ, etc., with any strategy for variable and value ordering. $\mathcal{F}$ does not, however, include the use of special purpose propagators, since without restrictions propagators can add arbitrary power to the algorithm.

Now add to $\mathcal{F}$ the ability to discover, store, and unit propagate standard nogoods; call this new family of algorithms $\mathcal{F}^{NG_{\rm stan}}$. In contrast, consider adding to $\mathcal{F}$ the ability to discover, store and unit propagate generalized nogoods; call this new family $\mathcal{F}^{NG_{\rm gen}}$.

Theorem 1

1. *For any CSP problem $\mathcal{P}$ and any algorithm $A \in \mathcal{F}$, we can add either s-nogood learning or g-nogood learning to A without increasing the number of nodes searched when solving $\mathcal{P}$.*
2. *There exists an infinite family of CSP problems of increasing size n on which any algorithm in $\mathcal{F}$ takes time that is a factor of size $O(2^n)$ larger than the run time of FCCBJ with either s-nogoods (FCCBJ+S) or g-nogoods (FCCBJ+G).*
3. *There exists an infinite family of CSP problems of increasing size n on which any algorithm in $\mathcal{F}^{NG_{\rm stan}}$ takes super-polynomial time (specifically $n^{\Omega(\log n)}$). In contrast forward checking, with a particular static variable and value ordering and simple deterministic strategy for learning g-nogoods, can solve these problems in time $O(n^2)$.*

Proof sketch:

1. It is not difficult to see that if we preserve the variable and value ordering, storing and propagating nogoods in a

backtracking algorithm can only decrease the size of the search tree.

2. This is a simple corollary of results proved in (Mitchell 2003) and (Beame, Kautz, & Sabharwal 2004). The argument is that without storing and reusing nogoods the run time of all algorithms in $\mathcal{F}$ is lower bounded by the minimum size of a restricted form of resolution refutation called a tree-resolution. Using stored nogoods removes this restriction allowing shorter refutations and smaller search trees. The result then follows from the fact that there are well know examples which have short non-tree resolution refutations but whose smallest tree-resolution is exponential in size.

3. The proof of the third point utilizes a family of problems given in (Mitchell 2003). By relating algorithms in the family $\mathcal{F}^{\mathcal{NG}_{\text{stan}}}$ to a restricted form of resolution called negative resolution (where every resolution step involves at least one clause containing only negative literals), and using a previous result from proof complexity theory (Goerdt 1992), Mitchell shows that any algorithm in $\mathcal{F}^{NG_{\text{stan}}}$ takes time at least $n^{\Omega(\log n)}$ on this family of CSP problems. Again the idea is that all algorithms in $\mathcal{F}^{NG_{\text{stan}}}$ have run times that are lower bounded by the minimum size of a negative resolution refutation. Generalized nogoods remove the restriction to negative resolution because g-nogoods can contain both positive and negative literals.

   In our proof we have shown that a simple g-nogood learning scheme and nogood storage (and thus reuse) allows forward checking to solve these problems in time $O(n^2)$.[3]

These results show that except for their runtime overhead, storing nogoods to prune future paths can only improve backtracking search, and can potentially yield exponential improvements. The last result shows that storing generalized nogoods can yield an additional super-polynomial improvement over standard nogoods (since g-nogoods subsume s-nogoods, g-nogoods can always simulate the performance of s-nogoods). It should be further noted that it is the storing of g-nogoods that adds extra power. If we utilize nogoods only for conflict directed backtracking and don't store them for future use (e.g., by removing line 11 of FCCBJ+S), then it can be shown that g-nogoods produce exactly the same backjumping behavior as s-nogoods.

As noted above, however, these theorems do not cover algorithms that utilize specialized propagators to achieve GAC. We turn to this issue next.

Nogoods from Global Constraints

Global constraints, i.e., constraints of large arity, are common in real CSPs. To deal with such constraints the field has developed the powerful notion of propagators. These are special purpose algorithms for efficiently enforcing local consistency, typically GAC, on global constraints.

[3] In (Mitchell 2003) it is shown that these problems can also be solved with 2-way branching using a different branching strategy. However, our proof shows that the key to Mitchell's result is not 2-way branching, but rather the ability to learn generalized nogoods.

To learn nogoods and utilize them for non-chronological backtracking, as we do in FCCBJ+S, we must have a nogood reason to label every pruned value. With forward checking values are pruned because of direct constraint violations, and thus the nogood reason for a pruned value is easy to identify. However, with higher levels of constraint propagation values can be pruned without any direct constraint violation.

With s-nogoods the standard technique for computing a nogood to label a value pruned by GAC on a constraint C is to compute the set of all decision assignments that contributed to the pruning of some value of some variable in $scope(C)$. This can be realized by taking as an initial g-nogood the assignment that is to be pruned along with all of the other pruned values among the domains of the variables in $scope(C)$. For example, if $C = X + Y < Z$ (with all domains being non-negative) and we have pruned $X \leftarrow 0$, then GAC will prune $Z \leftarrow 0$. The initial g-nogood then becomes $\{Z \leftarrow 0, X \not\leftarrow 0\}$: the pruned assignment and all other current pruned values. To obtain an s-nogood we then replace all non-assignments in this initial g-nogood by their nogood reasons (by inductive assumption these reasons are s-nogoods). Unfortunately, when C's arity is large, the resulting s-nogood will typically include all or almost all of the decision assignments.

It should be noted that these nogoods, just like the base constraint violating nogoods used in FCCBJ+S, are by themselves not useful. They cannot prune any search paths that would not be pruned by GAC. The initial nogood labeling a GAC pruning simply captures the conditions under which GAC pruned the value: under the same conditions GAC would perform the same pruning. The power of nogoods comes from learning new nogoods. These new nogoods move us beyond the power of the base level of constraint propagation by capturing subsets of the prefix that were not directly pruned by propagation but instead required some search to refute.

A nogood that contains the current set of decision assignments is called **saturated**. Saturated nogoods are not useful for backtracking (they force us to backstep to the previous level), nor are they useful for pruning future paths (the backtracking search will not examine this same setting of decision assignments again). Worse, saturated nogoods make all other nogoods learned along the same path saturated: these learned nogoods are the result of unioning other nogoods with the saturated nogood and this always yields another saturated nogood. Thus once a saturated or near saturated nogood is learned, nogood learning will be ineffective in the rest of the search along this path. This is the reason why nogood learning tends not to help GAC much when the standard technique for learning nogoods from GAC is used.

Generalized nogoods allow us to learn better nogoods from GAC propagators. As with s-nogoods, the initial g-nogood labeling a pruned value is not in itself useful, but these g-nogoods are more useful inputs for learning new nogoods, as we will demonstrate empirically in the next section. We use three different methods for generating g-nogoods from GAC in our experiments.

Method 1 generates a g-nogood reason for a GAC pruning by using the initial g-nogood described above. That

is, we take as the nogood reason for pruning $X \not\leftarrow a$ the assignment that is pruned ($X \leftarrow a$) along with all of the other pruned values among the domains of the variables in the scope of the constraint. This nogood is inexpensive to compute, and it outperforms the standard s-nogood that would be generated from it. However, for some constraints this g-nogood still tends to be overly long and not very useful for deriving new nogoods.

Method 2, which can be considerably more expensive to compute, is to find a set of value prunings that cover the supports of the pruned value. We take the pruned assignment $X \leftarrow a$, examine all of its supports in the constraint, and for each support pick a pruned value that caused that support to be lost. Since a pruned value might have caused multiple supports to be lost, greedy heuristics can be used in trying to construct a small set of pruned values that together cover all of $X \leftarrow a$'s supports. The nogood reason for $X \not\leftarrow a$ then becomes $X \leftarrow a$ and the computed set of support deleting non-assignments. Although this method yields better nogoods than the first, it becomes impractical when the arity of the constraint becomes large.

Method 3 generates g-nogoods from GAC by exploiting the special structure of the constraint. Thus, a special augmented propagator must be constructed for each constraint. It turns out that the paradigm of thinking about the set of pruned values that caused another value to be pruned can often lead to natural g-nogoods from propagators. We illustrate with the all-different constraint.

The all-diff constraint over the variable $V_1, \ldots, V_n$ asserts that each variable must be assigned a distinct value. All-different is perhaps the most well known global constraint. (Regin 1994) developed an algorithm for achieving GAC on an all-diff constraint that runs in $O(n^{2.5})$ time. The algorithm works by using bipartite graph matching to identify sets of variables $\mathcal{K}$, such that the number of variables in $\mathcal{K}$ is the same as the number of values in the union $\mathcal{D}$ of their domains. This means that in any solution, all values in $\mathcal{D}$ must be consumed by the variables in $\mathcal{K}$. As a consequence, all values in $\mathcal{D}$ can be removed from the domains of variables not in $\mathcal{K}$. A g-nogood for any pruned assignment is then easily seen to be the set of pruned values of variables in $\mathcal{K}$—it is because these values were pruned that $\mathcal{K}$ must consume all of $\mathcal{D}$. This set of pruned values is easy to compute by using information already computed by the propagator, and it can be a small subset of the total set of pruned values of variables in the scope of the all-diff (which would have been used by method 1).

Although we do not have space to further demonstrate, the same principle of looking for sets of pruned values can be used to obtain propagator specific g-nogoods from a number of other global constraints including *binary dot product* and *lexicographic ordering*.

Empirical Results

We conclude the paper with the results of an empirical evaluation of the methods described above. We have implemented a constraint solver that uses GAC, is able to learn either standard or generalized nogoods, and use these nogoods for intelligent backtracking (CBJ) and for pruning the future search via unit propagation of the stored nogoods. Our solver, called EFC (Katsirelos 2004) is publically available. All experiments were run on a 2.4 GHz Pentium IV with 3GB of memory. We tested the solver on various domains.

In presenting these results we utilize the following notation. Algorithms are named by the base name of the backtracking algorithm e.g., GAC. The suffix CBJ indicates that nogoods are used for conflict directed backtracking, the suffix +S indicates the addition of stored standard nogoods, and the suffix +G indicates the addition of stored generalized nogoods. For example, GACCBJ+G is GAC (maintain generalized arc consistency) augmented with nogoods for conflict directed backjumping and stored generalized nogoods; GACCBJ is GAC with nogoods used for conflict directed backjumping, but no storing of nogoods. When +S is used, the standard technique for computing an s-nogood for a GAC pruning is used, as described in the previous section. With +G, one of the three methods described in the previous section is used, and we will specify which method was used in which domain.

Logistics. The first domain is the logistics planning domain, adapted from the code used in (van Beek & Chen 1999). The baseline algorithm against which we compare is that used in van Beek & Chen, namely GACCBJ using the DVO heuristic `dom+deg` with a domain-specific tie-breaker. All constraints are propagated using GAC-Schema (Bessière & Régin 1997), a generic algorithm for enforcing GAC. We compare GACCBJ against GACCBJ+G using method 1 to learn g-nogoods from GAC. The first column shows the name of the instance, the second whether it was satisfiable or not. The third and fourth columns show cpu time and nodes visited, respectively, for the GACCBJ algorithm, while the fifth and sixth columns show CPU time and nodes visited for GACCBJ+G.

Problem	Sat?	GACCBJ		GACCBJ+G	
		Time	Nodes	Time	Nodes
10-11	UNSAT	-	-	**3906.26**	478861
15-15	SAT	52.4	89524	**2.29**	1861
18-11	SAT	497.34	155662	**89.09**	26943
22-11	UNSAT	85.17	10519	**13.85**	1573
26-12	UNSAT	678.55	99220	**19.4**	2704
26-13	UNSAT	-	-	**1899.09**	132881
28-12	UNSAT	-	-	**326.57**	38608
30-11	UNSAT	45.77	15338	**4.65**	2347

Table 2: Time and nodes visited for GACCBJ vs generalized nogood recording. A '-' means no solution was found after 20000 seconds.

There are a total of 81 problems in the suite. The instances presented in table 2 are the 8 most interesting ones. Of the rest, 71 instances were solved in under 10 seconds by both algorithms and 2 instances could not be solved by either algorithm within a timeout of 20000 seconds. We can see in this table that adding generalized nogoods (GACCBJ+G) yields a significant increase in performance. GACCBJ+G consistently outperforms GACCBJ on this suite. In fact,

there is no instance in this table where GACCBJ is faster than GACCBJ+G, even though it does happen on some of the easier instances.

GACCBJ+S did not yield any significant improvement over plain GACCBJ. We also tried learning standard nogoods for GAC prunings and generalized nogoods from conflicts on backtrack, again we obtain no significant improvement over plain GACCBJ. This means on this suite both improvements (learning generalized nogoods from conflicts on backtrack and from GAC) are necessary to achieve the full potential of generalized nogoods.

Crossword puzzles. The second domain is the crossword puzzle domain (Beacham *et al.* 2001). Here our baseline algorithm is the best algorithm from (Beacham *et al.* 2001), EACCBJ. EAC is a version of GAC that stores constraints extentionally, thus can be significantly faster than GAC-Schema at the expense of memory. We compare EACCBJ against EACCBJ+G using method 2 to learn g-nogoods from EAC. The results are shown in table 3. The first column shows the name of the instance. The second and third columns show cpu time and nodes visited, respectively, for the EACCBJ algorithm, while the fourth and fifth columns show CPU time and nodes visited for EACCBJ+G.

Problem	EACCBJ		EACCBJ+G	
	CPU	Nodes	CPU	Nodes
UK-21.01	106.91	1133	**68.79**	547
UK-21.10	117.44	812	**105.22**	799
UK-23.04	357.81	1709	**231.23**	1250
UK-23.06	448.33	1379	**404.42**	1293
UK-23.10	3343.96	7647	**2841.44**	6608
words-15.01	13547.55	550591	**183.75**	9625
words-15.02	157.92	4799	**107.67**	2907
words-15.04	788.95	37912	**225.41**	11090
words-15.06	8465.79	226022	**726.53**	19593
words-15.07	2308.69	61496	**1400.14**	35834
words-15.10	-	-	**4567.01**	84289
words-19.03	1217.11	50959	**167.54**	8934
words-19.04	-	-	**275.98**	9000
words-21.01	3727.49	64621	**2760.63**	55527
words-21.03	570.02	18066	**327.2**	8489
words-21.06	-	-	**188.81**	6567
words-21.09	117.57	3825	**56.93**	2411
words-21.10	-	-	**14878.77**	209914
words-23.02	115.96	8972	**27.48**	2928
words-23.03	2607.35	90843	**187.96**	10668
words-23.05	-	-	**4102.69**	128797

Table 3: Time and nodes visited for EACCBJ vs generalized nogood recording. A '-' means no solution was found after 20000 seconds.

The suite contains 100 instances, of which table 3 shows only those for which at least one algorithm needed 100 seconds to find a solution and at least one algorithm was able to find a solution within the timeout of 20000 seconds. We see that once again adding generalized nogoods significantly improved EACCBJ. EACCBJ+G dominates EACCBJ both in terms of CPU time as well as number of nodes visited. The improvements range from a few percentage points for the easier instances to more than two orders of magnitude for instances such as words-15.01 and words-21.06. Once again we tried using EACCBJ+S, but this yielded no significant improvement in performance, and when we learned standard nogoods from EAC, rather than generalized nogood (learning generalized nogoods only from conflicts) again our results were significantly worse than when both improvements were used.

Problem	GAC		GACCBJ+S		GACCBJ+G	
(w,g,s)	Time	Nodes	Time	Nodes	Time	Nodes
11-6-2	**0.23**	1776	0.63	1787	0.3	1758
13-7-2	**4.82**	18730	18.27	18746	6.2	18497
2-7-5	1586.44	35080126	218.04	242945	**4.39**	15755
2-8-5	-	-	1211.87	555325	**5.47**	16384
3-6-4	-	-	869.65	624084	**5.04**	13958
3-7-4	-	-	549.56	392693	**1.57**	7018
4-5-4	843.4	22525307	91.47	193181	**0.26**	2428
4-7-3	0.11	1455	0.25	1206	**0**	814
5-6-3	105.14	1973647	1142.57	592537	**1.3**	5896
5-8-3	218.86	1869855	277.9	207674	**50.31**	71541
9-8-4	**1.65**	2379	2.74	2379	2.02	2379

Table 4: Social golfer problem: CPU time and nodes visited for GAC vs GACCBJ with standard nogood recording and GACCBJ with generalized nogood recording, all with a static ordering. A '-' means no solution was found after 2000 seconds.

Social Golfer. The final domain in which we tested our algorithm is the social golfer problem (`prob010` in CSPLIB). We used the model outlined in (Frisch *et al.* 2002). Briefly, a three dimensional matrix of binary variables is used. Player i plays in week j in group k if the variable at position i, j, k is 1. Constraints are posted to ensure that each player plays in exactly one group each week, each group gets the correct number of players s, and each pair of players play each other at most once. In addition, lexicographic constraints (LEX) are posted between week planes, player planes and groups within each week to break symmetry. A static variable ordering that exploits the structure of the problem is used. We compare three algorithms: simple GAC using specialized propagators for all constraints, GACCBJ+S and GACCBJ+G. For GACCBJ+G we used method 3 (propagator specific g-nogoods) to learn g-nogoods from GAC, except for the LEX constraint where method 2 was used. The results are shown in table 4. The first column shows the instance name, the second and third CPU time and nodes visited for GAC, the fourth and fifth CPU time and nodes visited for GACCBJ+S and the sixth and seventh CPU time and nodes visited for GACCBJ+G.

When using stored nogoods, we found that the best results could be achieved when posting the decomposition of the LEX constraint as opposed to the global constraint. The reason for this is that even though propagator specific g-nogoods could be generated LEX (method 3), these g-nogoods were still too long. Even though we lose pruning power when using the decomposition, we avoid introducing long nogoods into the nogood database, while still retaining most of the symmetry breaking that LEX performs.

With this in mind, we see that nogood recording helps immensely in this domain. Even recording s-nogoods of-

fers close to an order of magnitude improvement in some instances, but it is not robust. Recording g-nogoods, on the other hand, is consistently faster, except for some very easy instances. On the hardest instances, it is as much as two orders of magnitude faster.

It should be noted that there exist results that supersede those presented here for the social golfer problem, using a different model and more advanced symmetry breaking techniques (Puget 2002). However, these symmetry breaking techniques are orthogonal to nogood recording, so can be applied in conjunction with it.

References

Bayardo, R. J., and Schrag, R. C. 1997. Using csp look-back techniques to solve real-world sat instances. In *Proc. of the AAAI National Conference*, 203–208.

Beacham, A.; Chen, X.; Sillito, J.; and van Beek, P. 2001. Constraint programming lessons learned from crossword puzzles. In *Proc. of the 14th Canadian Conference on AI*, 78–77.

Beame, P.; Kautz, H.; and Sabharwal, A. 2004. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research* 22:319–351.

Bessière, C., and Régin, J.-C. 1997. Arc consistency for general constraint networks: Preliminary results. In *Proceedings of IJCAI*, 398–404.

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

Focacci, F. and Milano, M. 2001. Global Cut Framework for Removing Symmetries. In *Proc. of Principles and Practice of Constraint Programming*.

Frisch, A.; Hnich, B.; Kiziltan, Z.; Miguel, I.; and Walsh, T. 2002. Global constraints for lexicographic orderings. In *Proc. of Principles and Practice of Constraint Programming*.

Goerdt, A. 1992. Unrestricted resolution versus n-resolution. *Theoretical Computer Science* 93(1):159–167.

Katsirelos, G. 2004. EFC constraint solver. *http://www.cs.toronto.edu/ gkatsi/efc/*.

Mitchell, D. 2003. Resolution and constraint satisfaction. In *Proc. of Principles and Practice of Constraint Programming*, 555–569.

Moskewicz, M.; Madigan, C.; Zhao, Y.; Zhang, L.; and Malik, S. 2001. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*.

Prosser, P. 1993. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence* 9(3).

Puget, J.-F. 2002. Symmetry breaking revisited. In *Proc. of Principles and Practice of Constraint Programming*.

Regin, J.-C. 1994. A filtering algorithm for constraints of difference in CSP. In *Proc. of the AAAI National Conference*, 362–367.

van Beek, P., and Chen, X. 1999. CPlan: A constraint programming approach to planning. In *Proc. of the AAAI National Conference*, 585–590.

Zhang, L.; Madigan, C.; Moskewicz, M.; and Malik, S. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *Proc. of IEEE/ACM International Conference on Computer Design (ICCAD)*, 279–285.