Unrestricted Nogood Recording in CSP search

George Katsirelos and Fahiem Bacchus

Department of Computer Science, University Of Toronto,*
Toronto, Ontario, Canada
[gkatsi,fbacchus]@cs.toronto.edu

Abstract. Nogood recording is a well known technique for improving backtracking search. However, in the field of Constraint Programming (CP) only restricted forms of nogood recording have been investigated, largely due to concerns about the time and space requirements of recording and using large numbers of nogoods. Recently, however, very successful SAT solvers have utilized the easy accessibility of gigabyte memories as well as more sophisticated data structures to demonstrate the effectiveness of recording gigabytes of long nogoods (clauses). In this paper we investigate the adaptation of these techniques to constraint satisfaction problems (CSPs), and provide some evidence that when properly adapted these techniques can improve our ability to solve CSP problems. We also investigate a generalization of standard nogoods that supports more powerful reasoning with the recorded nogoods.

1 Introduction

A number of works have investigated nogood recording as a technique for improving backtracking search, e.g., [FD94, Dec90]. Abstractly, a nogood is an easily checkable condition that can be used to test the nodes of a backtracking search tree. If the condition is true, then there can be no solution to the CSP below that node. If the condition is false the node remains feasible, and the subtree below it might, but need not, contain a solution. The idea behind recording nogoods is that as we explore nodes in the backtracking tree we discover various conditions (nogoods) such that when they hold no solution can exist. By recording these nogoods we can then test nodes newly visited by the search. If any of them make a previously recorded nogood true, we can immediately backtrack and thus avoid all further search below that node. Alternately we might discover that no solutions exist below the node even though the node does not satisfy any of the previously recorded nogoods. In this case we can record a new nogood. Thus nogoods are recorded incrementally and the future nodes are subject to more and more stringent feasibility conditions.

The difficulty with nogood recording is that an exponential number of nogoods, many very large, could potentially be recorded while solving a CSP. Large nogoods impose conditions that have typically been harder to check. Furthermore these conditions are also less frequently true; hence large nogoods can also be less useful for pruning the search space. The time and space requirements to process a large number of large nogoods have lead the CSP field to develop various restricted forms of nogood recording,

* This research was supported by the Canadian Government through their NSERC program.

where different criteria are used to limit the number and perhaps the size of nogoods that are recorded.

Nevertheless, in recent work on SAT solvers, [MMZ$^+$01], it has been found that gigabytes of clauses (which are identical to nogoods) can be effectively recorded and used when solving hard SAT problems. This work relies on two features to alleviate the space and time requirements of nogood recording: (1) the fact that machines with gigabyte memories have become commonplace; and (2) the development of data structures that significantly increase the efficiency of processing large numbers of large nogoods.

Gigabyte memories might seem to pale in the face of exponential numbers of nogoods, but it must be remembered that the number of nogoods stored is linear in the run time. Furthermore, as we store more and more nogoods, we can potentially prune more and more nodes in the remaining part of the search. Thus for many problems it becomes a race between memory usage and time—will the problem terminate prior to exhausting all available memory. Empirically, it would appear that gigabyte memories give just enough room for this race to be won on many interesting problems. Furthermore, even simple strategies like forgetting some number of the oldest or least recently used nogoods when memory has been exhausted seems to work well: having a large number of active nogoods during the search still seems to add significantly to improving efficiency.

In this paper we investigate the adaptation of SAT clause recording techniques to CSPs. There are two key differences between SAT and CSPs relevant to nogood recording. First, CSPs typically have non-binary variables, and second, in CSPs constraints are often represented intensionally as an algorithm for checking the constraint, rather than extensionally as a set assignments that satisfy the constraint. As long as one is dealing with variables with finite domains it is always possible to convert the CSP to a clausal SAT encoding. However, in doing so one looses the compactness of the CSP representation. In some cases this can cause an exponential blowup in the problem (e.g., an all-diff constraint over n variables), making the SAT encoding impossibly large. Furthermore, such a translation renders makes it much more difficult to apply the important practical device of special-purpose propagators. Hence the importance of finding ways of applying these techniques directly to the CSP encoding. Adapting these techniques to CSPs requires finding ways of dealing with non-binary variables and intensionally represented constraints efficiently.

In generalizing clause recording techniques from SAT it becomes apparent that nogoods, as standardly defined in CSPs, are not as general as they could be. For example, they do not support unit propagation which is an essential part of the effectiveness of recorded clauses in SAT. Part of our contribution in adapting clause recording techniques from SAT to CSPs is to define and investigate a natural generalization of nogoods that allows them to support the same kinds of reasoning as the recorded clauses in SAT.

Our empirical results provide evidence that unrestricted recording of nogoods can yield benefits when solving hard CSP problems. In particular, we demonstrate that these techniques can solve some problems that previously did not appear to be solvable with CSP backtracking algorithms. However, our results also show that the various clause recording techniques that are so effective in SAT, are not as uniformly effective in the CSP context. It remains an open question whether or not CSP specific modifications and improvements of these techniques can make them into more robust improvements of CSP algorithms.

2 Nogoods in Backtracking Search

A CSP consists of a set of variables $\{V_1, \ldots, V_k\}$ a domain of values for each variable, and a set of constraints. Each constraint can be viewed as being a boolean valued function over a subset of the variables. It maps each assignment of values to these variables to TRUE/FALSE. If its value is TRUE, we say that this particular assignment of values *satisfies* the constraint. A solution to the CSP is an assignment of a value to each variable such that every constraint is satisfied by this set of assignments.

The standard notion of nogood, as explored in the CSP literature, is *a set of assignments that cannot be extended to a solution of the problem.* Each node in the backtracking tree is defined by the set of assignments made so far, with all descendant nodes extending this set of assignments. Hence, if a node covers a nogood, i.e., includes all of the assignments in the nogood, all nodes in the subtree below it must also cover the nogood, and none of them can be a solution. So we can immediately backtrack from the node.

Nogoods are the CSP equivalent to the clauses learned by state-of-the-art SAT solvers, e.g., [MMZ$^+$01]. To see this consider the simplest SAT encoding of a CSP [Wal00]. We associate with each possible assignment of a variable, $V \leftarrow a$, a proposition asserting that V has been assigned that value. The constraints of the CSP are then encoded as clauses over this set of propositional symbols. In addition, the constraint that each variable must be assigned one and only value, implicit in the CSP encoding, is also encoded as a set of clauses. Any satisfying model of this SAT encoding will set the truth value of each of these "assignment" propositions, and the assignments comprising the CSP solution will be those propositions (assignments) that have been set to TRUE.

Under this "direct encoding", a nogood $V_1 \leftarrow a_1, \ldots, V_i \leftarrow a_i$ is equivalent to the clause $(V_1 \not\leftarrow a_1, \ldots V_i \not\leftarrow a_i)$. That is, at least one of the assignments (literals) in the nogood must be false. Rather than go back and forth between nogoods and the clauses they represent, we will talk strictly in terms of nogoods mapping the standard processing of clauses directly to the processing of nogoods.[1] Logically, a nogood is a conjunction of a set of literals all of which cannot be true, i.e., the conjunction must be false. If one of its literals becomes false the nogood is falsified and becomes inactive. If $k-1$ of its literals become true, the nogood is reduced to a unit nogood containing the k'th literal. This last literal is then forced to be false. Finally, if we have two nogoods with a complementary pair of literals, we can union them and remove the complementary pair. This is equivalent to resolving the two underlying clauses.

2.1 Nogood Processing

The manner in which nogoods are discovered and used during backtracking search is illustrated by the pseudo-code given in Table 1.

[1] Talking strictly in terms of clauses is more cumbersome since the algorithms are most easily understood as discovering nogoods.

```
NG-Rec(level)
1.  V = selectUnAssignedVar()
2.  if V == NONE
3.    print solution (current assignments) and exit
4.  foreach d ∈ UnPrunedDomain[V]
5.    assign(V,d)
6.    UnitPropNGStore()
7.    PropagateConstraints()
8.    BckLvl = NG-Rec(level+1)
9.    unassign(V,d)
10.   unPruneDomains(level)
11.   if BckLvl < level
12.     return
13. NG = ⋃{Nogood[V,d]-(V,d): d ∈ Domain[V]}
14. (BckVar,BckVal) = DeepestAsgn(NG)
15. BckLvl = LevelOf(BckVar,BckVal)
16. Nogood[BckVar,BckVal] = NG
17. pruneBckLvl = LevelOf(DeepestAsgn(NG-(BckVar,BckVal)))
18. pruneToLevel(BckVar,BckVal,pruneBckLvl)
19. AddToNGStore(NG)
20. return(BckLvl)
```

Table 1. Nogood Processing in Backtracking Search

After selecting the next variable V to branch on, the backtracking routine systematically attempts to assign it one of its currently unpruned values (line 4). After the assignment the nogood store is updated, to see if any previously discovered nogood has become unit. Any nogood NG that has become unit forces its remaining untrue assignment to be false. This can be accomplished by pruning the corresponding value from the domain of its variable. For example, if $V \leftarrow a$ is the last untrue assignment in NG, a will be pruned from the domain of V. NG becomes the nogood for that value (Nogood[V,a]=NG), and that value is marked as pruned at the current level.

Next constraint propagation is performed (line 7). Whenever a value a of a variable V is pruned by a constraint we set its Nogood[V,a] to be the set of assignments that caused the constraint violation. For example, if simple forward checking is being used, then a will be pruned when a constraint, C is violated. In this case the nogood will the assignments to the variables of C along with $V \leftarrow a$—this set of assignments violates the constraint. In more extensive forms of constraint propagation a value is pruned when its supports on a particular constraint are lost. In this case the nogood will be the union of the nogoods that caused these supports to be lost. It should be noted that the nogoods originating from constraint violations are never placed in the permanent the nogood store, they are only stored temporarily in the Nogood[] array.

After propagating both the constraints and the nogood store the search recursively examines the subtree below the current assignment (line 8). If this recursive call fails to find a solution it will either backtrack past this level, or it will assign a nogood for the current assignment by setting Nogood[V,d]. After examining the current assignment, all of the values that have been pruned to the current level are unpruned, and their Nogood[]'s emptied. Finally, after all unpruned values have been attempted, every value in `V`'s domain will have a nogood, and we can union these nogood together to obtain a new nogood `NG` (line 13).

The new nogood allows us to backtrack far enough so as to undo its deepest assignment (line 15), as well as prune that deepest assignment even further back so as not to violate the new nogood (line 17–18).[2] Finally, the new nogood is added to the nogood store to be used in the rest of the search. The new nogood is the result of composing a collection of nogoods, so it is implied by, but not directly encoded in the input constraints.

The recorded nogoods now act just like the original constraints. In particular, they cause domain values to be pruned and provide reasons for those prunings just like the original constraints. The difference is that an extensive search tree might have been explored to discover the nogood (i.e., the nogood might have been the result of many nogood unions over many backtracks). Hence, the nogood captures non-obvious consequences of the input constraints that can be very expensive to recompute.

The final issue to deal with is when a future variable has a domain wipe out (DWO), either through constraint propagation or through the propagation of unit nogoods. This is easily dealt with by forcing the variable selection heuristic (line 1) to always return a variable with an empty unpruned domain if one exists. In this case, the DWO will be immediately detected in the recursive call, in which the ***foreach*** loop of lines 4–12 will not be executed, a new nogood will be computed at line 13 (the code maintains the invariant that every pruned value has a `Nogood[]`, so each value of the DWO variable will have a corresponding nogood), and an appropriate backtrack computed.

This technique can also be used to deal with unit propagations of forced variables. That is, if we use a *unit preference* rule where we force the variable selection heuristic to always return a variable with only one remaining unpruned value if one exists (after first checking for variables with empty domains as above), then anytime a variable's value is forced via nogood or constraint propagation the search will immediately instantiate that variable. If that instantiation in turn forces other variables then they will be instantiated next. In this way chains of unit propagations will be realized by non-branching paths in the search tree.

The algorithm shown in Table 1 is essentially the same as already known techniques. Note that the input constraints are kept separate from the nogood store, and that nogoods that are derivable via constraint propagation from these constraints are not placed in the nogood store. Only non-obvious nogoods arising from resolving other nogoods are placed in the nogood store. Thus the nogood store does not duplicate the information contained in the input constraints, nor does it place any restriction on how the input constraints are represented.

Every node of the search tree generates a new nogood to be placed in the nogood store. Furthermore, these nogoods can potentially involve a large percentage of the assignments along the path to the node. Hence, they can be quite large at nodes deep in the tree. Since complex problems give rise to very large search trees, this raises concerns about the time and space required to record all of these nogoods. Frost and

[2] Pruning back of assignments we backtrack to (failed assignments) is explained in [Bac00]. Basically, we know that any attempt to make the failed assignment while the other assignments in the new nogood are still active is bound to fail.

Dechter [FD94] presented empirical evidence that the overheads were such that search was actually slowed down when all nogoods were recorded. To keep some of the benefits of nogood recording, but avoid its cost, Dechter [Dec90] proposed i^{th}-order length-bounded nogood recording, in which a nogood is only recorded (line 19) if it contains at most i assignments. The technique was extended to also record nogoods at internal nodes in [FD94] and to dynamic CSPs in [SV94]. Ginsberg [Gin93] proposed instead to record all nogoods, but discard them once they differ by more than one variable from the assignments in the current branch of the search tree. Bayardo and Miranker [BM96] generalized that proposal to i^{th}-order relevance bounded nogood recording, in which a nogood is discarded as soon as it differs from the current branch in i variables.

In this work we present two new results. First, we show how nogoods can be naturally generalized so that they can support the same kinds of reasoning as the clauses learned by SAT solvers. Second, we show how the data structures utilized in SAT solvers can be adapted so as to reduce to overhead of nogood recording. This reduction in overhead is sufficient to reverse Frost and Dechter's empirical results, showing that unrestricted nogood recording can in fact achieve superior results.

3 Generalizing Nogoods

In SAT solvers one of the main advantages of recording large numbers of clauses, is that the store of recorded clauses can generate long chains of unit propagations, quickly simplifying the problem. Nogoods however do not support such chains of unit propagation. In particular, all of the literals they contain have the same sign. Hence, a nogood can only be reduced in size via an assignment to a variable (which makes one of the literals in the nogood true, thus forcing one of the remaining literals to be false). When a nogood becomes unit it forces a domain value to be pruned. Constraint propagation also only forces domain prunings. Prunings can only make other nogoods inactive (by making one of their literals false); they can never reduce the size of a nogood. In fact, the only way that a chain of unit propagations can occur is via the unit preference rule described above.

The nogood store also does not support other forms of resolution based reasoning, since there are no conflicting literals that can be resolved. Finally, individual nogoods have a relatively low chance of becoming unit: the chance of a random assignment falsifying an ordinary k-length clause is $1/2^k$, but for variables with d values a random assignment makes a nogood true with probability only $1/d^k$, which decreases exponentially faster than $1/2^k$ with k. This can be confirmed empirically. We have found that only a very small number of the recorded nogoods are ever used to prune the future search.[3]

The solution to these difficulties is conceptually simple. All we need to do is to allow both assignments and non-assignments in nogoods. For example, say that we have the two nogoods $NG_1 = \{V_1 \not\leftarrow a, V_2 \not\leftarrow c, V_3 \leftarrow d\}$, and $NG_2 = \{V_1 \not\leftarrow b, V_3 \not\leftarrow d, V_4 \leftarrow c\}$; the search makes the assignment $V_1 \leftarrow c$; and the value c is pruned from the domain of V_2. $V_1 \leftarrow c$ implies that $V_1 \not\leftarrow a$ and $V_1 \not\leftarrow b$ both become true (V_1

[3] But which ones will be used is unpredictable which is why recording large numbers of them is effective: it increases the chance of some nogood being used.

can no longer be assigned either of these values). The pruning of c means that $V_2 \not\leftarrow c$ becomes true. Thus NG_1 is reduced to the unit nogood $\{V_3 \leftarrow d\}$, which is forced to be false by pruning d from the domain of V_3. This then causes NG_2 to become the unit $\{V_4 \leftarrow c\}$, resulting in c being pruned from the domain of V_4. This chain of unit propagations occurs without any variables being forced, and thus would not have occurred with standard nogoods and the unit preference rule.

Discovering Generalized Nogoods: To understand how general nogoods can be discovered during search, it is useful to consider why it is that the unioning of nogoods is a valid way of producing a new nogood. Implicit in the CSP representation is a "must have a value" nogood. If V has the values $x_1, \ldots, x_d$ in its domain, its "must have a value" nogood, M, will be $\{V \not\leftarrow x_1, \ldots, V \not\leftarrow x_d\}$. M asserts that it cannot be the case that V is not assigned x_1 and not assigned x_2, …, and not assigned x_d: it must be assigned one of these values. When we have a nogood for each of V's values, $NG_i = \{V \leftarrow x_i, R_i\}$, we can resolve NG_1 against M, and then NG_2 against the result, and so on until we resolve NG_d against the penultimate result. The final result will be $\{R_1, \ldots, R_d\}$—i.e., precisely the unioning of nogoods performed at line 13 of the nogood processing algorithm.

In order to discover general nogoods we keep track of the order in which the assignment and the non-assignment literals become true as we descend a path in the search tree. This sequence of literals is divided into levels, where the first literal made true at each node along the path marks the beginning of a level. These first literals are always assignment literals (an assignment to the variable selected at that node) and are called a *choice assignment*. The subsequent literals in the level are literals that are forced by constraint or nogood propagation (with general nogoods some of these literals might be forced assignments rather than forced prunings). For each of these forced literals we record the nogood that forced it.

As before, we backtrack whenever we have a nogood for every value of a variable V. However, instead of performing the above sequence of resolutions against the "must have a value" nogood, we start directly with the must have a value nogood. We know that each of the literals in it has become true, since all of V's values have been pruned at this point. We incrementally unwind this nogood always replacing the chronologically most recent literal by the reason (nogood) associated with it. The literals added from the reason must all have occurred chronologically before the replaced literal, so we are ensured monotonic progress on unwinding the nogood. Eventually, we have a nogood whose most recent literal is the first literal in its level, we can then backtrack to that level, undo the assignment, prune it and store the nogood as the reason for the pruning. The key is that during this incremental unwinding process some of the non-assignments in the "must have a value" nogood persist, and thus we discover nogoods with non-assignments.

For example, say that the following sequence of literals are made true during a descent of the search tree (with each literal numbered m.n with m being the level in the search tree the literal became true and n being the literal's chronological order within that level): (1.1) $V_1 \leftarrow a$, (1.2) $V_1 \not\leftarrow b$, (1.3) $V_1 \not\leftarrow c$, (1.4) $V_2 \not\leftarrow a$, (2.1) $V_3 \leftarrow a$, (2.2) $V_3 \not\leftarrow b$, (3.1) $V_4 \leftarrow a$, (3.2) $V_4 \not\leftarrow b$, (3.3) $V_2 \not\leftarrow b$, (3.4) $V_2 \not\leftarrow c$. At this point say that all of V_2's values have been pruned with associated reasons: ($V_1 \leftarrow a$, $V_2 \leftarrow a$),

$(V_3 \leftarrow a, V_4 \leftarrow a, V_2 \leftarrow b)$, and $(V_4 \leftarrow a, V_2 \leftarrow c)$. We start with the must have a value nogood $(V_2 \not\leftarrow a, V_2 \not\leftarrow b, V_2 \not\leftarrow c)$. The latest literal to become true was $V_2 \not\leftarrow c$ at level 3. Resolving this with the reason for the literal yields $(V_2 \not\leftarrow a, V_2 \not\leftarrow b, V_4 \leftarrow a)$, next is $V_2 \not\leftarrow b$ and resolving this away yields $(V_2 \not\leftarrow a, V_3 \leftarrow a, V_4 \leftarrow a)$. Now the most recent literal is the first literal of its level (a choice assignment), and we backtrack to undo the assignment $V_4 \leftarrow a$ at level 3, prune $V_4 \leftarrow a$ back to level 2, and post $(V_2 \not\leftarrow a, V_3 \leftarrow a, V_4 \leftarrow a)$ as the reason for $V_4 \not\leftarrow a$.

Notice that the discovered nogood has a non-assignment in it. It states that whenever we have $V_3 \leftarrow a$ and $V_2 \not\leftarrow a$ we cannot have $V_4 \leftarrow a$. If we had learned a standard nogood we would have produced the nogood $(V_1 \leftarrow a, V_3 \leftarrow a, V_4 \leftarrow a)$. This nogood only applies when $V_1 \leftarrow a$ (and $V_3 \leftarrow a$). But we know from the nogood $(V_1 \leftarrow a, V_2 \leftarrow a)$ that $V_1 \leftarrow a$ implies that $V_2 \not\leftarrow a$, so whenever it is applicable the more general one is also. Furthermore, there may be many other ways in which $V_2 \not\leftarrow a$ can become true, thus the general nogood can be applied in other situations as well.

Unique Implication Point (UIP): Notice that each of the resolution steps described above, where we replace the chronologically latest literal by its associated nogood, generates a new nogood. Any or all of these nogoods could be recorded in the nogood store.

SAT solvers that record all of the discovered nogoods (clauses) are typically slower [MSS99]. There are two more viable alternatives. The first, and standard technique, is to only record the nogood we obtain after we resolve back to a choice assignment. The second is to record the first nogood we obtain that contains only one literal made true at the current level. The nogood so produced is called a unique implication point (UIP), and in SAT, UIP clauses appear to particularly useful to record [ZMMM01]. It should be noted that with standard nogoods, UIP processing almost always produces the same nogood the standard technique. In particular, standard nogoods do not participate in chains of unit propagation, so the UIP in most cases resolves back to a choice assignment just like the standard technique. With generalized nogoods, however, nogoods computed via UIPs can be quite different from those computed via the standard technique. We have experimented with both types techniques for clause learning and report our results in Section 5.

Non-Binary Domain Processing The SAT encoding of a CSP contains explicit clauses to enforce the constraint that each variable must have a value, and can only have one value. These clauses (nogoods) could be added to the nogood store prior to search, but it is more efficient (and effective) to account for the implicit presence of these clauses by special processing of the nogoods. There are three cases that we process in this way. First, whenever a variable V is assigned a value a we make all literals $V \leftarrow x$ for $x \neq a$ false.[4]. Second, if a nogood contains an assignment to a variable we remove all non-assignment to the same variable prior to storing the nogood. For example, we reduce

[4] The implicit nogood is the binary "only one value" $(V \leftarrow a, V \leftarrow x)$. This nogood would become unit when $V \leftarrow a$ is made true, forcing the literal $V \leftarrow x$ to be false.

the nogood $\{V \leftarrow a, V \not\leftarrow b, R\}$ to $\{V \leftarrow a, R\}$.[5] Third, if a nogood is reduced to a collection of non-assignments to the same variable, we immediately prune all other values from the variable's domain. For example, if we have reduced a nogood to the set $(V \not\leftarrow a, V \not\leftarrow b, V \not\leftarrow d)$, where the domain of V is $\{a, b, c, d, e\}$, then we can immediately prune c and e from the domain of V (the original, unreduced, nogood is set as the reason for these prunings). [6] It can be noted that the last two cases cannot captured by simply unit propagating the implicit "must have a value" and "only-one-value" nogoods.

Chains of Unit Propagations Generalized nogoods are used in the same way as standard nogoods—we detect when they become unit and then force the remaining literal to be false by either making an assignment or by pruning a value. With simple nogoods we can perform one pass of nogood propagation (line 6) followed by one pass of constraint propagation (line 7)—values pruned by constraint propagation could not cause more nogoods to become unit. With general nogoods, however, we have to iterate lines 6 and 7, until no more literals are forced—now values pruned by constraint propagation can make nogoods unit, and thus we have to go back and propagate those nogoods. These newly propagated nogoods can in turn cause values to be pruned which might cause additional constraint propagation, etc. Eventually these propagations stop and we continue with the backtracking search.

If these propagations cause a domain wipe out, the wiped out variable will be selected in the next recursion (as described above), a new nogood computed and the search backtracked as before. However, the propagations might also cause all of the literals in a nogood to become true. This also is a failure, and we can halt all further propagations. The failed nogood can then be chronologically unwound in precisely the same manner as the union of the domain nogoods of a wiped out variable. Thus, the algorithm can backtrack from this type of failure in the same manner as an exhausted variable domain.

4 Efficient Implementation

The second contribution of our work is simply to point out that the lazy data structures utilized by SAT solvers can be applied to manage the nogood store. The resulting gain in efficiency makes unrestricted nogood recording more effective than previous investigated restricted versions.

The main functionality we need from the nogood store is to be able to efficiently detect when a nogood becomes unit. A literal in a nogood might become true via a variety of different mechanisms. It might be set true when a choice assignment is made (note that the assignment $V \leftarrow a$ makes all other literals of the form $V \not\leftarrow x$, for $x \neq a$ true), when constraint propagation prunes a value, or when it is forced by a nogood

[5] This corresponds to a Krom-subsumption step, where we resolve the nogood with the "only-one value" $\{V \leftarrow a, V \leftarrow b\}$. The result subsumes the original nogood, and hence can replace it.

[6] This corresponds to a Hyper-Binary resolution step [Bac02] where we resolve the nogood against a set of "only-one value" nogoods. For example, we resolve $(V \not\leftarrow a, V \not\leftarrow b, V \not\leftarrow d)$ against $(V \leftarrow e, V \leftarrow a)$, $(V \leftarrow e, V \leftarrow b)$, and $(V \leftarrow e, V \leftarrow d)$, to derive $V \not\leftarrow e$.

becoming unit. Nevertheless, for the purpose of detecting new unit nogoods, all that the nogood store needs to know is the set of literals that have newly become true.

This functionality is exactly the same as is required in a SAT solver, which must detect when a clause becomes unit given that some set of literals have newly become true. The simplest way of implementing this functionality is to associate a count and a falsified flag with every nogood. The count represents the number of literals contained in the nogood that have not been set to true. The falsified flag is set if any literal in the nogood has been made false, indicating that the nogood is no longer active. Whenever a literal is made true during the search, all of the unfalsified nogoods containing its negation get their falsified flag set, while the unfalsified nogoods containing it get their count decremented. If this count becomes one, we know that the nogood now contains only one literal that is not true. The nogood is then searched to find the remaining untrue literal, which must then be made false by either pruning a value or forcing an assignment.

This simple approach, however, is very time consuming when we have large numbers of long nogoods. In particular, whenever we prune or assign a value we have to visit every nogood containing it, and when we backtrack to we have to do just as much work to restore the count and falsified flags. Long nogoods make for more work as they contain many assignments. so they increase the average number of clauses a value appears and thus the work require to process an assignment or pruning.

Recently developed techniques in SAT solvers for efficiently detecting unit clauses [Zha97, MMZ$^+$01] have demonstrated that the simple approach described above can be greatly improved. The idea is to introduce two *watch literals* in each nogood. The nogood cannot become unit unless at least one of these watch literals becomes true. When a literal becomes true we examine all of the nogoods it is a watch for (a fraction of the nogoods it appears in). Each of these nogoods is searched to try to find another literal (other than the other watch) that is false or as yet unset. If we find one we make it a new watch literal for the nogood, replacing the old, now true, watch literal. If we fail to find such a literal, we know that the nogood has become unit, and that the other watch literal must be forced to be false. On backtrack we need not do anything, the new watch literals remain valid watches on backtrack. Furthermore, since each nogood has the same number of watch literals, the average number of nogoods watched by a literal (and thus the amount of work we need to do when processing a literal) is immune to the length of the nogoods, it just depends on the number of nogoods. The technique of watch literals significantly improves the efficiency of unrestricted nogood recording.

5 Empirical Results

We have implemented unrestricted nogood recording, with standard nogoods as well as generalized nogoods. We have also implemented UIP processing which can be used with generalized nogoods. In SAT solvers, UIP processing seems to be more successful when combined with a “variable decay” heuristic [MMZ$^+$01]. We experimented with a simple version of that heuristic, but were unable to achieve good results. Hence, we leave further exploration of such heuristics for future work. All experiments were per-

	EAC		FCCBJ+RB		FCCBJ+S		FCCBJ+G		FCCBJ+UIP	
Problem	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
UK-21.04	20.36	447	817.36	5924008	1491.22	1170356	-	-	5312.95	824067
UK-23.06	84.72	1306	-	-	5203.07	2994762	-	-	1028.48	835313
UK-23.10	210.54	1834	177.06	1236956	134.59	415718	2199.28	710566	-	-
words-15.01	-	-	-	-	5395.93	2423058	-	-	1122.5	655479
words-15.10	2.29	265	-	-	5752.77	3702947	-	-	-	-
words-19.03	502.12	21096	-	-	156.56	526797	-	-	490.26	482251
words-19.04	10.58	580	-	-	15.46	118325	43.47	124524	30.53	97721
words-21.01	-	-	-	-	3921.84	5056685	-	-	1266.06	1207743
words-21.06	4.35	484	-	-	45.11	168548	350.27	287223	31.94	75443
words-23.03	-	-	-	-	144.57	672178	5006	1902648	1959.23	1400760
words-23.04	207.16	3783	-	-	14715.87	6194196	-	-	-	-
words-23.08	-	-	-	-	-	-	-	-	4529.65	1817350
words-23.09	269.79	9933	-	-	-	-	-	-	-	-

Table 2. Crossword puzzles with FC-based algorithms

formed on a Pentium-4 2.2 GHz machine, with 4GB of RAM, and times are reported in CPU seconds.

Our first set of experiments utilized hard crossword puzzles. In [BCSvB01], Beacham et al. performed an extensive study of different algorithms, models, and heuristics for solving 50 different crossword layouts using two different dictionaries, “UK” and “words”. They identified an optimized version of MGAC (maintain generalized arc consistency), a particular model of the problem, and a particular variable ordering heuristic, whose combination provided the best performance on these problems (among the *thirty four* different combinations they experimented with). They called this algorithm/model/heuristic combination EAC.[7]

In our experiments we compared Beacham et al.’s implementation of EAC against various nogood recording algorithms. We found that forward checking was the appropriate underlying algorithm upon which to run the various forms of nogood recording in order to obtain the best performance. Most of the 100 different problems (50 layouts times 2 dictionaries) could be solved all of the algorithms with no major differences in the time required. The other problems, i.e., those where at least one algorithm failed (with a 20,000 second time bound) are presented in Table 2.

Besides showing the performance of EAC on these problems we also show the performance of the following nogood recording algorithms (all of which utilize forward checking in their constraint propagation phase and conflict directed backtracking [Pro93] to support the discovery and utilization of the nogoods).

FCCBJ+RB This algorithm implements 3^{rd} order relevance bounded recording of standard nogoods.

[7] Interestingly, EAC does not use any specialized propagators. Nevertheless, it was faster than algorithms that did make extensive use of propagators.

FCCBJ+S This algorithm implements unrestricted recording of standard nogoods. In all cases where unrestricted recording is utilized the nogood store is pruned by removing the oldest nogoods if we ever run out of space.

FCCBJ+G This algorithm implements unrestricted recording of generalized nogoods. It uses the standard technique of unwinding conflicts to the deepest choice assignment when learning nogoods.

FCCBJ+UIP This algorithm implements unrestricted recording of generalized nogoods. However, it utilizes UIP processing of the conflicts when learning nogoods.

We had also experimented with length-bounded nogood recording, but it was not competitive with any of the other algorithms, so we omit those results.

We see that relevance bounded nogood recording is not very effective, always being slower than EAC, and solving fewer problems. FCCBJ+S, on the other hand, is a significant improvement over EAC in terms of its ability to solve problems (and thus it is also a significant improvement over all of the other algorithms experimented with in [BCSvB01]). FCCBJ+S takes significantly more time than EAC on many problems, but it is able to solve more problems than any other algorithm (four more than EAC). EAC does, however, solve words-23.09, which no other algorithm is able to solve.

FCCBJ+G, where the UIP technique is not used, does not perform well on this problem suite. It is not as powerful as FCCBJ+S, nor EAC. It does solve problems that relevance bounded nogood recording cannot solve, but FCCBJ+S solves all of the problems it can solve and many more. FCCBJ+UIP, however, does demonstrate some of the advantages of generalized nogoods. In most cases it is an improvement over FCCBJ+G, solving more problems and solving them faster than FCCBJ+G. It is also often faster than FCCBJ+S and solves a number of problems that EAC cannot. It is also able to solve one problem, words-23.08, that no other solver is able to solve.

It is also interesting to examine in more detail the nogood processing performed by the nogood recording algorithms. Some information about this is presented in Table 3. These problems are representative of the general case. Learning generalized nogoods is more efficient in the sense that fewer nogoods are accumulated (fewer nodes are visited). Furthermore, these nogoods are more frequently applicable (they generate more prunings). Hence, in a fundamental sense, generalized nogoods are more powerful than standard nogoods. However, we also see that they are propagated more frequently (propagations occur when a nogood's watch literal becomes false). Hence, they entail a higher overhead. Standard nogoods are only propagated when assignments are made. Generalized nogoods, however, can be propagated by domain prunings. In a typical search, many more domain prunings will occur than assignments.

For the crossword problems this added overhead is not always compensated for by the reduction in the number of nodes searched. The problem UK-21.04 is illustrative. On this problem FCCBJ+UIP prunes almost 8 million more domain values than does FCCBJ+S. Yet this yields a less than 200,000 node savings in the size of the search space.

In the next set of problems, we use problems from the logistics planning domain, modeled as CSPs [vBC99]. In these problems, the best algorithm to date has been GAC-CBJ (see also [CvB01] for a more detailed comparison of different algorithms). Again most of these problems were easy for all of the algorithms. The results for the more

	FCCBJ+S			FCCBJ+UIP		
Problem	# Nogoods	# Propagations	# Prunings	# Nogoods	# Propagations	# Prunings
UK-21.04	595,066	1,120,958,704	3,003,420	382,395	1,394,158,274	11,981,624
UK-23.06	1,600,997	18,250,022	4,742,048	374,261	313,170,705	6,338,429
words-15.01	1,353,056	3,927,473,719	5,487,747	262,494	430,312,608	7,026,755
words-19.03	278,315	89,278,090	511,613	211,375	179,182,583	4,270,734
words-21.01	2,412,605	2,851,534,352	3,789,663	448,911	489,044,329	8,397,413

Table 3. Nogood activity in crossword puzzles

	GACCBJ		GACCBJ+S		GACCBJ+G	
Problem	Time	Nodes	Time	Nodes	Time	Nodes
15-15	21.6	13137	11.46	5089	11.53	4569
18-11	490.00	94504	494.04	94504	496.29	92584
19-11	0.68	933	0.75	933	0.75	933
20-13	10.71	4509	2.35	844	2.94	971
22-11	640.00	23538	212.98	9618	44.2	1707
25-14	2.84	1156	3.06	1156	3.1	1156

Table 4. Logistics planning problems

interesting problems are shown in Table 4. On these problems we found that GAC (generalized arc consistency), rather than FC, was the best underlying algorithm upon which to run the various forms of nogood recording in order to obtain the best performance. In the table we report GACCBJ, GACCBJ+S (unrestricted recording of standard nogoods), and GACCBJ+G (unrestricted recording of generalized nogoods using the standard unwinding technique to learn nogoods from conflicts). The data is presented in Table 4.

We also experimented with length and relevance bounded nogood recording, but in both cases these produced little or no change over the times taken by simple GACCBJ. Surprisingly, on these problems GACCBJ+UIP, where UIP processing is used to learn nogoods from conflicts, proved to be very poor on this suite of problems. In particular, it failed to solve many problems that were easily solvable by the rest of the algorithms.

On these problems we see that GACCBJ+S is an improvement over simple GACCBJ. The improvement is not great on most problems. But it never performs much worse than GACCBJ, and on problem 22-11 it provides a good speedup. GACCBJ+G performs even better, offering a significant speedup on 22-11, and again never being much slower than either of GACCBJ or GACCBJ+S.

More details about the nogood processing on some representative problems is presented in Table 5. Here we see that, unlike the crossword puzzles, the number of propagations of the generalized nogoods is not very different from the number of propagations of the standard nogoods (once we account for the number of nodes searched by the two algorithms). However, the generalized nogoods continue to have an advantage in terms of the average number of times each nogood is used. This helps explain why

	FCCBJ+S			FCCBJ+UIP		
Problem	# Nogoods	# Propagations	# Prunings	# Nogoods	# Propagations	# Prunings
15-15	1,219	27,987	152	1,056	24,768	483
18-11	23,844	15,093,842	2	23,484	18,346,866	10,622
20-13	190	1,074	62	232	1,769	283
22-11	3,212	524,197	3,600	272	24,561	1,151

Table 5. Nogood activity in planning problems

generalized nogoods perform well on these problems. What remains an open question is to explain the poor performance of UIP processing on this problem suite.

6 Conclusion

We have presented a method for discovering, recording and using unrestricted nogoods in backtracking search, along with some evidence as to the usefulness of such nogoods. We note that unrestricted nogood recording does allow us to solve some problems that could not previously be solved by other backtracking search techniques. We have also presented a natural generalization of nogoods that makes them much more like the clauses recorded by SAT solvers, and shown how to adapt other SAT techniques, like UIP, that on some problems they can be useful.

Overall, we have found that the techniques utilized with such positive effects in SAT solvers, are not as uniformly effective when translated to CSPs. Nevertheless, these techniques can occasionally be effective. This leads to a number of natural questions involving trying to understand, and predict, when these techniques can be effective for CSPs. Furthermore, it could be that CSP specific modifications or improvements of these SAT techniques, e.g., better heuristics, might allow them to be more robustly effective.

References

[Bac00] Fahiem Bacchus. Extending forward checking. In *International Conference on Principles and Practice of Constraint Programming*, number 1894 in Lecture Notes in Computer Science, pages 35–51. Springer-Verlag, New York, 2000.

[Bac02] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Proceedings of the AAAI National Conference*, pages 613–619, 2002.

[BCSvB01] Adam Beacham, Xinguang Chen, Jonathan Sillito, and Peter van Beek. Constraint programming lessons learned from crossword puzzles. In *Proceedings of the 14th Canadian Conference on Artificial Intelligence*, pages 78–87, 2001.

[BM96] R. J. Bayardo Jr and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 298–304, Portland, Oregon, 1996.

[CvB01] X. Chen and P. van Beek. Conflict-directed backjumping revisited. *JAIR*, 14:53–81, 2001.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[FD94] Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of the AAAI National Conference*, pages 294–300, 1994.

[Gin93] M. L. Ginsberg. Dynamic backtracking. *J. of Artificial Intelligence Research*, 1:25–46, 1993.

[MMZ+01] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of the Design Automation Conference (DAC)*, 2001.

[MSS99] J. P. Marques-Silva and K. A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[Pro93] P. Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence*, 9(3), 1993.

[SV94] T. Schiex and G. Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.

[vBC99] P. van Beek and X. Chen. CPlan: A constraint programming approach to planning. In *Proceedings of the AAAI National Conference*, pages 585–590, Orlando, Florida, 1999.

[Wal00] Toby Walsh. Sat v csp. In *International Conference on Principles and Practice of Constraint Programming*, number 1894 in Lecture Notes in Computer Science, pages 441–456. Springer-Verlag, New York, 2000.

[Zha97] H. Zhang. Sato: An efficient propositional prover. In *Proceedings of the Fourteenth International Conference on Automated Deduction (CADE)*, volume 1249 of *LNCS*, pages 272–275. Springer-Verlag, 1997.

[ZMMM01] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of IEEE/ACM International Conference on Computer Design (ICCAD)*, pages 279–285, 2001.