Learning implied constraints lazily

Neil Moore and Ian Miguel

School of Computer Science, University of St Andrews, St Andrews, Scotland
{ncam,ianm}@cs.st-andrews.ac.uk

Abstract Explanations are a technique for reasoning about constraint propagation, which have been applied in many learning, backjumping and user interaction algorithms. To date, explanations have been recorded "eagerly" when the propagation happens, which leads to inefficient use of time and space, because many will never be used. In this paper we show that it is possible and highly effective to calculate explanations retrospectively when they are needed. To this end, we implement "lazy" explanations in a state of the art learning framework. Experimental results confirm the effectiveness of the technique: we achieve reduction in the number of explanations calculated up to a factor of 200 and robust reductions in overall solve time up to a factor of 2.

1 Introduction

Typically, CSP solvers use backtracking search to find solutions to constraint problems. Usually search is supplemented by propagation; however, most forms of propagation are incomplete because they cannot generally find solutions or detect dead ends without the aid of search. Adding implied constraints to the problem can reduce the incompleteness of propagation, by allowing more values to be removed that cannot be in any solution.

This paper describes improvements to the technique of producing *explanations* for what propagators do, which in turn improves Katsirelos' g-nogood learning [15] and other algorithms that use explanations. Our main contributions are as follows:

- Introduce the idea of lazy explanations, which to reduce the overhead of propagation in learning frameworks and other algorithms where explanations are needed.
- Show how to implement the technique in a state of the art learning solver.
- Demonstrate improvement in CSP learning technology by up to a factor of two decrease in overall solve time on a large set of benchmark problems, as well as showing that number of explanations computed are universally decreased up to a factor of 200 reduction.

We finish by suggesting future directions for research.

For example, the constraint occurrence$([v_1, \ldots, v_n], 2, 4)$ ensures that exactly 4 of the variables $v_1, \ldots, v_n$ are assigned to 2. If $v_1, \ldots, v_4$ are all assigned to

Algorithm 1 (left) standard search algorithm (right) search with learning

```
Standard-Search()
if ∀x ∈ Var, |d(x)| = 1
    output solution
    exit
choose an unset variable x to branch on
for v ∈ dom(x)
    post branching constraint that x ← v
    propagate all constraints
    if ¬∃x ∈ Var s.t. |d(x)| = 0
        StandardSearch()
    retract x ← v
return
```

```
Learning-Search()
if ∀x ∈ Var, |d(x)| = 1
    output solution
    exit
choose an unset variable x to branch on
for v ∈ dom(x)
    post branching constraint that x ← v
    propagate all constraints
    if ∃ true g-nogood
        analyseConflict()
        retract x ← v
    else
        target← Learning-Search()
        retract x ← v
        if target ≠ x
            return target
    propagate all constraints
    if ∃ true g-nogood
        target←analyseConflict()
        return target
```

2, then 2 can be pruned from $dom(v_5)$ and $dom(v_6)$ and so on. Some algorithms like CBJ [20] and g-learning [15] rely on *explanations* being available for prunings; in the example the explanation for 2 being pruned from v_5 is $\{v_1 \leftarrow 2, v_2 \leftarrow 2, v_3 \leftarrow 2, v_4 \leftarrow 2\}$ because these 4 assignments are sufficient for the pruning. In numerous previous papers, such explanations have been built *eagerly* at propagation time. In this paper we observe that many are never needed and hence we describe how to evaluate them *lazily* and show the benefit of doing so.

2 Explanations and nogoods

The most notable and up to date algorithm that uses explanations is Katsirelos et al's [15,16,17] g-nogood learning (g-learning). For this reason we will use g-nogood learning as a framework in which to present our progress with explanations. Unless alternative citation is given, all material in this review section is based on that work. Towards the end of this section we give historical context describing other explanation-based algorithms for the CSP.

We describe the g-learning scheme by contrasting it with a "standard" search solver (Algorithm 1 (left), which can be characterised as d-way branching search with propagation, variable ordering heuristics and chronological backtracking). g-nogood learning is Algorithm 1 (right).

The first significant way that learning differs from a standard solver is that whenever a propagator assigns or prunes a value it must store a *g-nogood* as an explanation for the action.

Definition 1. *A* g-nogood *is a set of assignments and prunings that cannot all be true in any solution to the CSP.*

If a propagator causes pruning $x \nleftarrow a$ then it must store a g-nogood of the form $\{x \leftarrow a, d_1, \ldots, d_k\}$ (where d_i are (dis-)assignments) such that $d_1, \ldots, d_k$ becoming true is sufficient for the pruning. For example if the propagator for $x \neq y$ causes $x \nleftarrow a$ because $y \leftarrow a$ it will label the pruning with $\{x \leftarrow a, y \leftarrow a\}$. Here $y \leftarrow a$ is an *explanation* for the pruning $x \leftarrow a$, since if ever $y \leftarrow a$ happened again we could certainly prune a from x. Another way of looking at this is that nogood $\{x \leftarrow a, y \leftarrow a\}$ is an implied constraint that the propagator subsumes, hence if $y \leftarrow a$ then the propagator *must* do $x \nleftarrow a$ to ensure the constraint is satisfied.

g-nogoods must be stored for all assignments and prunings, except decision assignments, which are labelled with NULL to denote that they are unconnected with other decisions and inferences. Values pruned as a result of an assignment to a variable are labelled by an "at most one value" (AMOV) nogood, e.g., if $w \leftarrow a$ then prunings $w \nleftarrow k$ where $k \neq a$ will be labelled by $\{w \leftarrow a, w \leftarrow k\}$. Finally, an assignment $y \leftarrow a$ caused by the elimination of all but one value for y would be labelled by the "must have a value" (MHAV) nogood: $\{y \leftarrow \min(dom(y)), \ldots, y \leftarrow \max(dom(y))\}$. Note that all g-nogoods used as explanations are implied constraints in the CSP.

A nogood can be propagated using unit propagation, i.e., when all but one of the (dis-)assignments are true, make the remaining one false to ensure the CSP can still be solved.

Next, learning differs from the standard solver because a depth is stored for every assignment and pruning in the format $d.s$ where d is the decision depth and s is a sequence number within the depth, e.g., 2.0 for a decision at depth 2 and 0.7 for the seventh assignment or pruning at the root node (depth 0).

The final difference between learning and the standard solver is the way that conflicts are handled. Learning assumes that the nogood store has been preloaded with a MHAV g-nogood for every variable, so that domain wipeouts (DWOs) are handled by a failure in the nogood store. Rather than backtracking, a conflict analysis procedure will run (analyseConflicts()) and this is when the g-nogood labels are exploited. The aim is to obtain a new g-nogood that is added to the constraint store after backtracking, to prevent similar conflicts occuring again. The algorithm used is to begin with the failed g-nogood and to repeatedly resolve it with the deepest assignment or dis-assignment's nogood, so that the (dis-)assignment is removed and replaced by its causes. This continues until a termination condition is reached, usually the firstUIP condition [26] that the nogood contains exactly one (dis-)assignment from the depth at which failure occurs. Now the solver backtracks and the g-nogood is added. The details of this backtracking are beyond the scope of this paper, however the pseudocode

shows the analyseConflict() procedure returning a jump target which the solver returns to before continuing search.

A more detailed discussion of the correctness and completeness of the g-nogood learning algorithm is, unfortunately, beyond the scope of this paper, however it is important to emphasise certain essential properties of the g-nogoods used to label (dis-)assignments. Suppose a nogood $\{v \leftarrow a, d_1, \ldots, d_k\}$ labels pruning $v \not\leftarrow a$:

Property 1. At least one of $d_1, \ldots, d_k$ must have become true at the same decision depth as $v \not\leftarrow a$ occurred.

Remark 1. The property is true of GAC propagators, for example, but not bounds consistency Z propagators [2]. In a proof of correctness of g-learning it ensures that a firstUIP cut exists.

Property 2. None of $d_1, \ldots, d_k$ may have a depth greater than or equal to $v \not\leftarrow a$.

Remark 2. Ensures that causes must precede effects and avoids cycles in the g-learning implication graph.

Section 4 describes an improvement to this algorithm: working out g-nogoods lazily as they are needed, instead of eagerly as the propagations are done. As we will show empirically in Section 5, many nogoods are never needed in the process of building implied constraints.

We now review earlier research involving learning and/or explanations, to show that explanations are common in CSP algorithms and to convince the reader that lazy explanations are a new idea. The earliest CP-specific work was by Frost and Dechter [5,4] on value- and graph-based learning; and jump-back learning. Value- and graph-based begin with the complete failing instantiation (which is a nogood) and remove assignments that are not needed for it to be a nogood. A precomputed table is used to establish if a value is compatible with all values in another variable, rather than explanations. The jump-back scheme piggy-backs on conflict-directed backjumping (CBJ) [20,21]. CBJ collects explanations eagerly so it can later work out the reasons for failures. Ginsberg's dynamic backtracking [10] builds "eliminating explanations" eagerly to provide knowledge of which assignments were the cause of inconsistent values in other variables. Schiex et al. [24] describe how to build and use generic nogoods that are not made by the propagator. Jussein et al. [23,12,13] described how to produce explanations (consisting of assignments only) for global constraints for various purposes including integrating MAC and dynamic backtracking, user interaction and learning constraints. They were produced eagerly by propagators. Cambazard et al. used eagerly built explanations for variable and value ordering heuristics. G-learning is a significant improvement on previous learning schemes as it makes the insight that g-nogoods, i.e., nogoods containing dis-assignments as well as assignments, are far superior in terms of compactness and propagation power.

3 Lazy Versus Eager Learning

We will now give an description of the execution of an eager g-nogood learning CSP solver, and contrast its operation with that of our proposed lazy approach. The CSP we will use has:

- variables v, w, x, y and z, each with a starting domain of $\{1, \ldots, 5\}$; and
- constraints alldiff(v, w, x, y, z) and $x = z$.

This CSP is unsatisfiable, as it is easy to prove by searching using the variable ordering $[x, z, \ldots]$. Nevertheless[1] we will use the variable ordering $[v, w, x, y, z]$ to demonstrate thrashing being avoided using learning and illustrate the main techniques.

The following paragraph is shown in detail in Table 1. Observe that no GAC propagation is possible at the root node. The search process begins by making the assignment $v \leftarrow 1$. This causes the remaining values $2, \ldots, 5$ to be removed from v in the normal way. At this point standard eager g-learning will store a nogood for each pruning (specifically the AMOV nogood $v \leftarrow 1, v \leftarrow i$ for each $i \in 1, \ldots, 5$). In contrast, lazy learning stores just a pointer to the responsible constraint (AMOV). Next, alldiff removes the value 1 from the domains of all the other variables in its scope (w, x, y and z). Again g-nogood labels will be stored for each, whereas lazy learning stores just a pointer to alldiff. Now propagation cannot progress any further. Another decision to assign $w \leftarrow 2$ is made; similar propagation results. The next decision to assign $x \leftarrow 3$ causes $x = z$ to propagate, setting $z \leftarrow 3$. However alldiff has also forced $x \not\leftarrow 3$. Hence a contradiction has been derived, because z cannot be both 3 and not 3.

A conventional solver now backtracks and takes a right branch, whereas our learning solver backtracks and learns a new constraint. The constraint is intended to stop the same conflict from happening again, and in due course these learned constraints can be combined to remove entire failing subtrees of the search tree.

Eager learning begins with the *nogood* $\{z \not\leftarrow 3, z \leftarrow 3\}$. This nogood can be annotated by the depth at which each of the constraints became true: $\{z \not\leftarrow 3@3.4, z \leftarrow 3@3.5\}$. It is first resolved with the stored label for the deepest literal, $z \leftarrow 3$ to obtain $\{z \not\leftarrow 3@3.4, x \leftarrow 3@3.0\}$. Now the g-nogood is resolved again, this time with the label for $z \not\leftarrow 3$ to obtain $\{z \leftarrow 3@3.0\}$.

Using lazy explanations, the starting nogood and resolutions are identical but lazy learning does not fetch nogood labels from storage. Rather, a polymorphic function of the constraint is invoked with the (dis-)assignment as a parameter. Based on the propagator's semantics a label can be built retrospectively. For example, if the constraint alldiff$([v, w, x, y, z])$ is asked why it pruned $2 \in x$ then it can easily work out that it was because $w \leftarrow 2$. Other cases are more complex but we shall show in Section 4 that it is always possible and advantageous to be lazy in this way.

[1] bear in mind that it has been shown [15] that on some problems learning will provide a superpolynomial speedup irrespective of the variable ordering used

CSP constraints: alldiff($[v, w, x, y, z]$), $x = z$				
Depth	Action	Constraint	Description	g-nogood label
$v \in \{1, \ldots, 5\}, w \in \{1, \ldots, 5\}, x \in \{1, \ldots, 5\}, y \in \{1, \ldots, 5\}, z \in \{1, \ldots, 5\}$				
1.0	$v \leftarrow 1$	null	search decision	null
1.1	$v \not\leftarrow 2$	AMOV	remove other value from decision variable	$\{v \leftarrow 2, \underline{v \leftarrow 1}\}$
1.2	$v \not\leftarrow 3$	AMOV	“	$\{v \leftarrow 3, \underline{v \leftarrow 1}\}$
1.3	$v \not\leftarrow 4$	AMOV	“	$\{v \leftarrow 4, \underline{v \leftarrow 1}\}$
1.4	$v \not\leftarrow 5$	AMOV	“	$\{v \leftarrow 5, \underline{v \leftarrow 1}\}$
1.5	$w \not\leftarrow 1$	alldiff	alldiff removes value that's already used	$\{w \leftarrow 1, \underline{v \leftarrow 1}\}$
1.6	$x \not\leftarrow 1$	alldiff	“	$\{x \leftarrow 1, \underline{v \leftarrow 1}\}$
1.7	$y \not\leftarrow 1$	alldiff	“	$\{y \leftarrow 1, \underline{v \leftarrow 1}\}$
1.8	$z \not\leftarrow 1$	alldiff	“	$\{z \leftarrow 1, \underline{v \leftarrow 1}\}$
$v \in \{1\}, w \in \{2, \ldots, 5\}, x \in \{2, \ldots, 5\}, y \in \{2, \ldots, 5\}, z \in \{2, \ldots, 5\}$				
2.0	$w \leftarrow 2$	null	search decision	null
2.1	$w \not\leftarrow 3$	AMOV	remove other value from decision variable	$\{w \leftarrow 3, \underline{w \leftarrow 2}\}$
2.2	$w \not\leftarrow 4$	AMOV	”	$\{w \leftarrow 4, \underline{w \leftarrow 2}\}$
2.3	$w \not\leftarrow 5$	AMOV	”	$\{w \leftarrow 5, \underline{w \leftarrow 2}\}$
2.4	$x \not\leftarrow 2$	alldiff	alldiff removes value that's already used	$\{x \leftarrow 2, \underline{w \leftarrow 2}\}$
2.5	$y \not\leftarrow 2$	alldiff	”	$\{y \leftarrow 2, \underline{w \leftarrow 2}\}$
2.6	$z \not\leftarrow 2$	alldiff	”	$\{z \leftarrow 2, \underline{w \leftarrow 2}\}$
$v \in \{1\}, w \in \{2\}, x \in \{3, 4, 5\}, y \in \{3, 4, 5\}, z \in \{3, 4, 5\}$				
3.0	$x \leftarrow 3$	null	search decision	null
3.1	$x \not\leftarrow 4$	AMOV	remove other value from decision variable	$\{x \leftarrow 4, \underline{x \leftarrow 3}\}$
3.2	$x \not\leftarrow 5$	AMOV	”	$\{x \leftarrow 5, \underline{x \leftarrow 3}\}$
3.3	$y \not\leftarrow 3$	alldiff	alldiff removes value that's already used	$\{y \leftarrow 3, \underline{x \leftarrow 3}\}$
3.4	$z \not\leftarrow 3$	alldiff	”	$\{z \leftarrow 3, \underline{x \leftarrow 3}\}$
3.5	$z \leftarrow 3$	$x = z$	must assign z to same as x	$\{z \not\leftarrow 3, \underline{x \leftarrow 3}\}$
conflict, because $3 \in z$ is both assigned and pruned				
$v \in \{1\}, w \in \{2\}, x \in \{3\}, y \in \{3, 4, 5\}, z \in \emptyset$				

Table 1. Trace of learning algorithm, in the "g-nogood label" column the explanation part is underlined

The nogood, however derived, can now be posted into the solver and it is globally false, i.e., the nogood is false in all solutions to the CSP. Hence the final action is to backtrack once to remove the incorrect branching decision, learn the nogood and continue.

Lazy explanations are intended to reduce the amount of overhead that learning places on propagators. Previously techniques like learning generalised nogoods and CBJ[20] required data to be collected during search giving reasons for each pruning, however now we need only store a pointer to a piece of code that is able to work out the reason retrospectively. This brings learning CP solvers more in line with SAT solvers, which also need only store a single pointer per propagation to enable learning [18].

4 Lazy learning

Conventionally a propagator will store a set of (dis-)assignments eagerly whenever a pruning is done. An alternative is to store only enough data to allow the nogood to be reconstructed efficiently later in the *same branch of the search tree*, this we call *laziness.*

4.1 Explanations for clauses

If clause $(x \nleftarrow a \vee d_1 \vee \ldots \vee d_k)$ causes pruning $x \nleftarrow a$, it is sufficient to note only which constraint did it, i.e., to store a pointer to the clause. We know that the pruning was by unit propagation[3] and at that point all of $d_1, \ldots, d_k$ were false. Hence the nogood is $\{x \leftarrow a\} \cup \{\neg d_1, \ldots, \neg d_k\}$ or informally the negative of the clause itself.

This form of lazy learning is very familiar because it is what SAT solvers do [18]. It is natural for SAT solvers to do lazy learning, but we will show that it is also possible and advantageous for CP solvers.

4.2 Explanations for inequalities

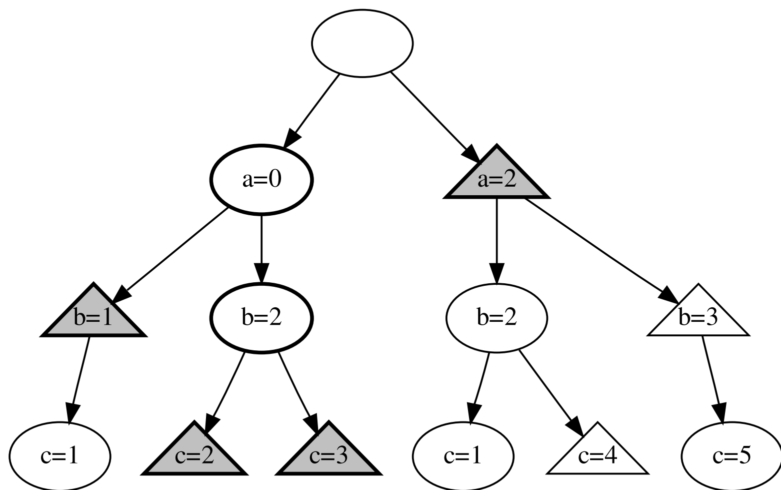
Suppose that constraint $x < y$ causes pruning $x \nleftarrow a$; it is sufficient to store only a pointer to the constraint $x < y$ to later reconstruct the nogood label. a is pruned if and only if all values in y greater than a are removed. Hence nogood label $\{x \leftarrow a\} \cup \{y \nleftarrow a+1, \ldots, y \nleftarrow \max(\text{initdom}(y)\}$ can be computed when required.

The ability to create nogoods lazily is only intended to be available later in the *same branch*, because later in the branch we can implement domain tests in $O(1)$ time for earlier states, as we will now illustrate by showing how to produce lazy explanations for the table propagator.

4.3 Explanations for table

Assume we are using a watched literals (WL) [6] implementation of table where tuples are stored in an array of tries, one per variable, so that all tuples involving a particular variable and value (varval) are readily accessible, introduced by [7] and illustrated in Figure 1. For example, the trie at the top of Figure 1 could represent every tuple involving $d = 1$, meaning that the set of tuples is $\{(d = 1, a = 0, b = 1, c = 1), (d = 1, a = 0, b = 2, c = 2), (d = 1, a = 0, b = 2, c = 3), (d = 1, a = 2, b = 2, c = 1), (d = 1, a = 2, b = 2, c = 4), (d = 1, a = 2, b = 3, c = 5)\}$. The propagator works by making sure that each varval is supported by at least one tuple, if any component of that tuple is lost either a new supporting tuple can be found in the trie, or the varval is pruned.

Such a constraint will prune a varval if and only if every tuple involving that varval has has at least one component pruned. Hence an explanation for the pruning is a set that includes *at least* one pruned component per tuple. We



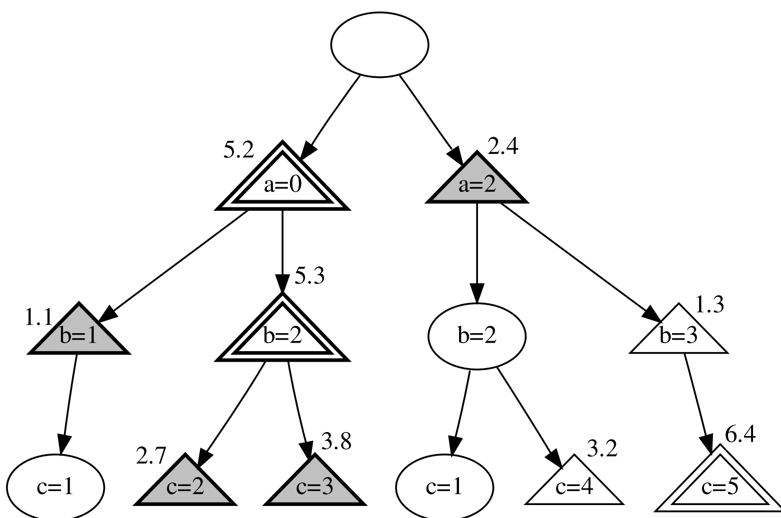


Figure 1. (top) Trie with pruned values shown as triangles, greyed nodes are those included in the nogood and nodes visited in the traversal are bold. (bottom) Same trie but values pruned between the original pruning (at depth 3.9) and the nogood being produced are in double triangles. Pruning depths are shown: permissible prunings have depth < 3.9, disallowed prunings have depth > 3.9.

demonstrate explanations for GAC-schema using Katsirelos' naïve scheme [15] which was arguably the most successful of the techniques he tried. The algorithm simply picks any pruned component from each tuple.

This can easily be implemented with tries: perform an inorder traverse of the trie but whenever a pruned node is visited add the node to the set and don't recurse any further. Each pruning covers all the tuples beneath the point in the tree when it was added. The top of Figure 1 illustrates this process.

Lazily, we are presented with the same trie, but with *at least* as many pruned values. We cannot be sure of satisfying Property 2 by applying the same traversal, for later additional prunings could be wrongly used when they could have had no effect on the earlier propagation. Instead, we adapt the algorithm to add to the set only values that were pruned *at that time*; this we achieve by inspecting the depth of the pruning. Such a situation is illustrated in the bottom of Figure 1 where we choose not to use the double lined triangular nodes where the original algorithm would have done.

A nogood could be built eagerly with no increase in asymptotic time complexity since the WL propagators must traverse the entire tree anyway prior to doing any pruning. By being lazy we incur at most one extra trie traversal per nogood because we might have to repeat the traversal when the nogood is requested. However fewer traversals will be needed overall if fewer than half of the explanations are needed.

The previous examples illustrate that lazy nogood generation can be just as efficient as eager evaluation, but with the additional advantage that it may never become necessary. The next example will show that lazy explanations can be efficient even for complex propagators like GAC alldiff. And, of course, explanations can be still done eagerly with no loss of efficiency when it is hard to work out nogoods retrospectively for a particular constraint.

4.4 Explanations for alldifferent

The alldifferent constraint (see [8] for a review) ensures that the variables in its scope take distinct values. For example, consider the variable value graph in Figure 2, where we have 4 variables and 5 values. The current domains are illustrated by having an edge from variable v to value a whenever $a \in dom(v)$.

Régin's GAC alldifferent propagation algorithm [22] first creates a maximum matching (size 4 matching shown with bold lines in the figure) in $O(r^{1.5}d)$ time and then uses Tarjan's algorithm to find Hall sets in $O(rd)$ time. Hall sets are sets of k variables such that the union of their current domains has size k (we refer to this union as the *combined domain*). Clearly the variables in a Hall set must consume the combined domain and so the values can be removed from the domains of all other variables. In the figure an unsupported value $2 \in dom(z)$ is shown with a dotted line, it is unsupported because 2 is used by the Hall set $\{w, x, y\}$.

To enable explanations to be produced later, a pointer to the constraint is stored for each pruning. Later, an explanation can be produced as follows:

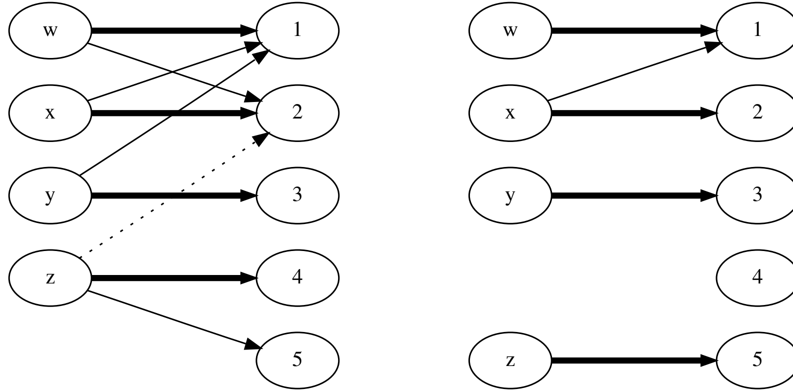


Figure 2. (left)Variable value graph at time of original pruning (right) Same graph at time of explanation

1. The alldiff propagator maintains a maximum matching as domains are narrowed. The most recent complete matching would have been valid when the pruning was done, since edges are only ever removed as domains are narrowed. Notice that the matching in Figure 2 (right) is also valid for Figure 2 (left).
2. Find the Hall set that consumed the pruned value earlier, by running Tarjan's algorithm, but using earlier domain state reconstructed by inspecting pruning depths, as described in Subsection 4.3.
3. The explanation is the conjunction of all the prunings from the Hall set (except the values in the combined domain), since the removal of these values ensured that the Hall set HS's combined domains consisted of $|HS|$ values. This operation is $O(rd)$.

Hence, in the example of Figure 2 the explanation is $\{w \not\leftarrow 4, w \not\leftarrow 5, x \not\leftarrow 4, x \not\leftarrow 5, y \not\leftarrow 4, y \not\leftarrow 5\}$. These prunings are enough to ensure only 3 values remain in $\{x, y, z\}$'s combined domain.

Contrast this with eager explanations, where the Hall set is known when the pruning is done, and the explanation can then be built in $O(rd)$ time. Hence, when we consider prunings individually, lazy learning's worst case time complexity of $O(rd)$ matches the eager approach, with the additional advantage that some of them will never be built. Note that there is an additional advantage for eagerness, which is that the same explanation could be used for several values and hence built only once; lazily it may be built several times. This means laziness is theoretically worse if the number is prunings per propagation is not bounded by a constant. It is not clear which variant will win in practice.

4.5 Explanations for arbitrary propagators

We have now described how to apply the lazy approach to a variety of constraints. Katsirelos' GAC-Generic-Nogood [15] is a procedure for finding g-nogood labels for an arbitrary propagator with unknown implementation: the explanation of a (dis-)assignment is just the set of prunings from other variables in the scope of the constraint. It can easily be evaluated lazily by including only prunings that were made before the propagation happened, a similar trick to Sections 4.3 and 4.4. In this way we can be sure that a generic nogood can always be produced lazily, although by specialising for each propagator as described above we will obtain smaller nogoods and/or reduce the time taken to compute them.

5 Experiments

We evaluated the effectiveness of lazy explanations in a g-learning solver.

5.1 Implementation

Our g-learning solver is based on minion public repository version 1885[2]. We make implementation decisions so that as far as possible we are changing only the method used for generating nogoods, compared to the experiments in [15]. Hence we choose to implement our solver with d-way branching, dom/wdeg variable ordering [1] and far backtracking as described in [15]. Our solver tries to use a firstUIP cut, but in case the firstUIP doesn't propagate the firstDecision cut is tried next and must cause a pruning. We believe that Katsirelos' solver uses firstDecision once a loop is detected but the details are unpublished [14]. Finally node counts are not directly comparable because we do not know how they were calculated.

Our learning implementation stores (dis-)assignment depths in an array. To produce explanations we store an small object with a polymorphic function that produces an explanation. For eager, the stored nogood is returned immediately; for lazy, the function implements an algorithm to calculate the nogood. In our implementation nogoods are not memoized, hence they may be calculated multiple times. Learned clauses are propagated by the 2-watch literal scheme [19].

5.2 Benchmarks

We used a large and varied set of benchmarks, consisting of:

- crossword problems,
- antichain problem,
- peg solitaire instances, and
- every extensional instance from the 2006 CSP Solver Competition.

[2] svn co https://minion.svn.sourceforge.net/svnroot/minion/trunk minion

With the exception of antichain, these were all produced by Tailor [9] using instances from the CSPXML repository[3] and those described in [11]. We chose these instances because we have to date implemented lazy explanations for constraints $=$, $\neq$, $<$, literal, not literal, disjunction (of arbitrary constraint), table and negative table.

5.3 Experimental methodology

Each of the 1418 instances was executed three times with a 10 minute timeout, on a Red Hat Linux server kernel 2.6.18-92.1.13.el5xen with 8 Intel Xeon E5430 cores at 2.66 GHz. Each run was identical, and we use the minimum time for each in our analysis, in order to approximate the run time in perfect conditions (i.e., with no system noise) as closely as possible. Each instance was run on its own core, each with 996MB of memory. Minion was compiled statically (`-static`) using g++ with flag `-O3`.

5.4 Results - lazy learning vs. no learning

First we will briefly give some results comparing lazy learning with no learning at all, i.e., ordinary minion with d-way branching. Figure 3 shows that learning is effective on certain classes of benchmark, but more work remains to be done to make it robust across a larger range of benchmarks. Some of these results differ from Katsirelos' [15]. This is because minion is a very highly optimised solver (it explores a much greater number of nodes per second) and hence in order to compensate for the overhead of learning a larger reduction in nodes is required. However, we do not know of faster solution times for peg solitaire [11] and other classes achieve speedups of up to 100x.

5.5 Results - lazy learning vs. eager learning

Now we turn our attentions to the subject of this paper: are lazy explanations effective in reducing the runtime of the g-learning framework? The answer is yes. Figure 4 shows an overall reduction in number of explanations generated in all cases, up to a factor of 200 reduction. This proves that the rationale behind lazy learning is correct—many explanations are never used and hence we should try to avoid calculating them.

Next we exhibit Figure 5, which confirms that, on the whole, time is saved by using lazy explanations: lazy explanations can double our solver's search speed, without affecting the search tree traversed significantly[4]. Note that this speedup is the whole solver, not just the learning engine. In other solvers where learning is less of an overhead the speed increase may be less, but we have been careful to optimise both lazy and eager learning in our solver. We carried out a non-parametric t-test (Wilcoxon signed ranked test [25]) and found that the difference between lazy and eager is statistically significant at the 1% level.

[3] http://www.cril.univ-artois.fr/ lecoutre/research/benchmarks/benchmarks.html

[4] sometimes lazy and eager make different choices between suitable nogoods

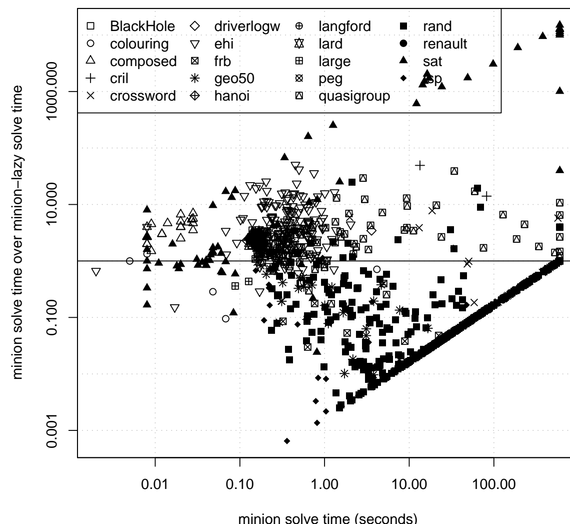


Figure 3. Scatterplot showing runtime comparison for minion versus minion-lazy. Each point is a result for a single instance. The x-axis is the solve time (i.e., excluding set up time which is identical for both). The y-axis gives the speedup from using minion-lazy instead of minion. A ratio of 1 means they were the same, above 1 means minion-lazy was faster and below 1 that minion was faster. Subsequent graphs are the same style.

Lazy learning is detrimental to a small number of instances. Note that the SAT instances which are below the line are probably noise, because their runtime is very small and furthermore for SAT clauses lazy and eager learning are the same. The quasigroup instances below the line are interesting: Figure 4 shows that most explanations are eventually used in the learning process for these instances. The increase is search time reflects the fact that lazy explanations for the table and negative table constraints require additional traversals of the trie data structure compared with eagerness (see Section 4.3).

6 Conclusions and Future Work

We have introduced *lazy explanations*, in which nogoods are computed as needed, rather than stored eagerly. This approach conveys the twin advantages, confirmed experimentally, of reducing storage requirements and avoiding wasted effort for nogoods that are never used.

In future work, we will create lazy explainers for constraints other than those featured herein. A further important item of future work is to investigate for specific propagators whether laziness is advantageous.

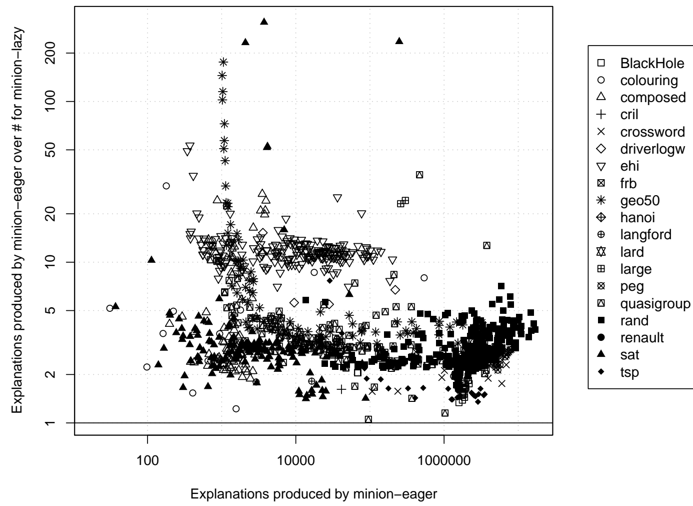


Figure 4. Scatterplot showing comparison of number of explanations produced by minion-lazy versus minion-eager, fewer for instances above the line.

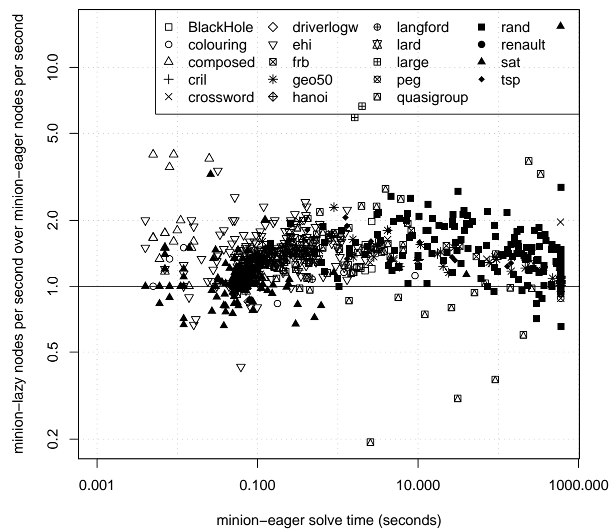


Figure 5. Scatterplot comparing nodes per second for minion-lazy versus minion-eager, more for instances above the line.

Acknowledgements

Thanks to George Katsirelos for invaluable discussions about g-learning. Thanks to Ian Gent, Lars Kotthoff, Pete Nightingale and Andrea Rendl for helping with the preparation of the paper and experiments. Thanks to Chris Jefferson his assistance with coding and algorithms.

The authors are supported by ESPRC grant number EP/E030394/1 - "Watched Literals and Learning for Constraint Programming".

References

1. F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *Proceedings of the 16th European Conference on Artificial Intelligence (ECAI'04)*, pages 482–486, Valencia, Spain, August 2004.
2. Chiu Wo Choi, Warwick Harvey, Jimmy H.-M. Lee, and Peter J. Stuckey. Finite domain bounds consistency revisited. In *Australian Conference on Artificial Intelligence*, pages 49–58, 2006.
3. Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.
4. Rina Dechter. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. *Artif. Intell.*, 41(3):273–312, 1990.
5. Daniel Frost and Rina Dechter. Dead-end driven learning. In *Proceedings of the Twelfth National Conference of Artificial Intelligence (AAAI-94)*, volume 1, pages 294–300, Seattle, Washington, USA, July 31 - August 4 1994. AAAI Press.
6. Ian P. Gent, Christopher Jefferson, and Ian Miguel. Watched literals for constraint propagation in minion. In *CP*, pages 182–197, 2006.
7. Ian P. Gent, Christopher Jefferson, Ian Miguel, and Peter Nightingale. Data structures for generalised arc consistency for extensional constraints. In *AAAI*, pages 191–197, 2007.
8. Ian P. Gent, Ian Miguel, and Peter Nightingale. Generalised arc consistency for the alldifferent constraint: An empirical survey. *Artif. Intell.*, 172(18):1973–2000, 2008.
9. Ian P. Gent, Ian Miguel, and Andrea Rendl. Tailoring solver-independent constraint models: A case study with essence' and minion. In *SARA*, pages 184–199, 2007.
10. Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.
11. Christopher Jefferson, Angela Miguel, Ian Miguel, and S. Armagan Tarim. Modelling and solving english peg solitaire. *Comput. Oper. Res.*, 33(10):2935–2959, 2006.
12. Narendra Jussien and Vincent Barichard. The PaLM system: explanation-based constraint programming. In *Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000*, pages 118–133, Singapore, September 2000.
13. Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

14. George Katsirelos, December 2008. Personal correspondence.
15. George Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, unpublished.
16. George Katsirelos and Fahiem Bacchus. Unrestricted nogood recording in csp search. In *CP*, pages 873–877, 2003.
17. George Katsirelos and Fahiem Bacchus. Generalized nogoods in csps. In Manuela M. Veloso and Subbarao Kambhampati, editors, *AAAI*, pages 390–396. AAAI Press / The MIT Press, 2005.
18. J. P. Marques-Silva and K. A. Sakallah. Grasp: A new search algorithm for satisfiability. In *International Conference on Computer-Aided Design*, pages 220–227. ACM Press, November 1996.
19. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.
20. Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence, Volume 9, Number 3*, pages 268–299, 1993.
21. Patrick Prosser. MAC-CBJ: maintaining arc consistency with conflict-directed backjumping. Technical Report Research Report/95/177, Dept. of Computer Science, University of Strathclyde, 1995.
22. Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94: Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.
23. Guillaume Rochart, Narendra Jussien, and Francois Laburthe. Challenging explanations for global constraints. In *CP03 Workshop on User-Interaction in Constraint Satisfaction (UICS'03)*, pages 31–43, Kinsale, Ireland, 2003.
24. Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problem. *International Journal of Artificial Intelligence Tools*, 3(2):187–207, 1994.
25. Frank Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.
26. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.