Lazy explanations for constraint propagators

Ian P. Gent, Ian Miguel and Neil C.A. Moore

School of Computer Science, University of St Andrews, St Andrews, Scotland
{ipg,ianm,ncam}@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 for constraint programming. To date explanations for constraints have usually been recorded "eagerly" when constraint 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 reductions in overall solve time up to a factor of 5.

Key words: constraint programming, explanations, learning

1 Introduction

Constraints are a powerful and natural means of knowledge representation and inference in many areas of industry and academia. Consider, for example, the production of a university timetable. This problem's constraints include: the maths lecture theatre has a capacity of 100 students; art history lectures require a venue with a slide projector; and no student can attend two lectures simultaneously. Constraint solving of a combinatorial problem proceeds in two phases. First, the problem is modelled as a set of decision variables, and a set of constraints on those variables that a solution must satisfy. In our example one might have two decision variables per lecture, representing the time and the venue. For each class of students, the time variables of the lectures they attend may have an alldiff constraint on them to ensure that the class is not timetabled to be in two places at once. The second phase consists of using a constraint solver to search for solutions: assignments of values to decision variables satisfying all constraints.

Typically, constraint solvers use backtracking search supplemented by constraint propagation, which is a form of inference. Propagation usually involves removing domain values that cannot be involved in any solution. This can dramatically reduce the space of assignments searched. Search can be further improved by the use of constraint learning where previously unknown constraints are uncovered during search and used to speed up search subsequently. Discovering these new constraints requires reasoning about *why* propagation removes values, which is why we need *explanations* for what it does.

This paper describes improvements to existing techniques for producing explanations, which in turn improves Katsirelos' g-nogood learning [15, 16, 17] and other CSP algorithms that use explanations. Our main contributions are as follows:

- To introduce the idea of lazy explanations for constraints, similar to a successful idea from satisfiability modulo theories (SMT) solvers. To our knowledge this technique has never been applied to the CSP before. The technique reduces the time and space overhead of propagation by calculating explanations only when they are needed.
- To show how to implement the technique in a state of the art learning solver.
- To describe for the first time how to produce explanations for various common global constraints lazily (i.e., only when needed). Currently the SMT community are incorporating constraint propagation algorithms into their tools (see SAT 09 invited talk [22]), so these new algorithms can be incorporated into SMT solvers as well as CSP solvers. We also prove that laziness can be implemented for any propagator by providing a lazy generic explanation algorithm.
- To 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.

We finish by describing related work and suggesting future directions for research.

2 Background: Constraints, search and explanations

It is necessary for us to provide some background describing the constraint satisfaction problem (CSP), CSP solvers and explanations in this section.

2.1 CSP and CSP solvers

A CSP is a triple (V, D, C) where V is the sequence $(v_1, \ldots, v_n)$ of *variables*, D is the sequence $(d_1, \ldots, d_n)$ of finite *domains*, where $\forall i,\ d_i \subset \mathbb{Z}$, and C is the set $\{c_1, \ldots, c_e\}$ of *constraints*. Each constraint c_i is over a subset $\{v_{c_1}, \ldots, v_{c_k}\}$ of the variables (the constraint's *scope*) and the allowed combinations of values are specified by a relation $R_i \subseteq d_{c_1} \times \ldots \times d_{c_k}$. However, usually a constraint will be specified in intension, i.e., the relation is implicit in the definition of the constraint. When a constraint c in included in C, we say that c is *posted*.

Usually, the aim is to find one or more *solutions* to the CSP, each of which is an assignment to all of the variables from their respective domains, such that the values in the scope of each constraint form an allowed combination (*satisfy* the constraint).

Our reference search solver in this paper can be characterised as depth first search with propagation, ordering heuristics and chronological backtracking. Hence the solver repeatedly assigns a variable v_i to a value $v \in d_i$, we call these branching decisions *decision assignments*. After each value is assigned, *constraint propagation* is carried out, whereby values that cannot be in any solution are removed:

Example 1. If constraint $v_2 \neq v_3$ is posted and v_2 is assigned to 3 then propagation will remove 3 from d_3, since assigning 3 to v_3 will result in failure.

The propagation procedure is repeated to a fixpoint. Now provided that no inconsistency has been discovered (i.e., a domain with no possible values) search will proceed to assign another variable, otherwise search will *backtrack* by retracting the most recent decision and continuing. Once a complete assignment is reached a solution has been found.

We use the notation $v_i \leftarrow a$ as a shorthand for "v_i assigned to a", i.e., all other values are removed from d_i. Similarly $v_i \not\leftarrow a$ for a *pruning* (alternatively, *disassignment*) where domain d_i loses value a.

A *propagator* is an implementation of a particular constraint; roughly, it must not prune any value that can be part of a satisfying assignment for the constraint. A propagator usually prunes according to a defined *level of consistency*. The most common one is *generalised arc consistency* (GAC) [19]. GAC propagation ensures that for every variable v_i and value $a \in d_i$ there is an assignment to the scope of the constraint that satisfies the constraint and assigns $v_i \leftarrow a$. If the variable/value pair cannot be part of such an assignment it is pruned.

2.2 Explanations

One of the most notable and up to date CSP algorithms that uses explanations is Katsirelos *et al*'s [15, 16, 17] g-nogood learning (g-learning). For this reason we will use g-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.

We describe the g-learning scheme by contrasting it with the standard solver described in the previous section. The first significant way that a learning solver differs is that whenever a propagator assigns or prunes a value it must store an *explanation* for the action:

Definition 1. *An* explanation for pruning $x \not\leftarrow a$ *is a set of assignments and disassignments that are sufficient for a propagator to infer* $x \not\leftarrow a$*. Similarly an* explanation for assignment $y \leftarrow b$ *is a set of (dis-)assignments that are sufficient for a propagator to infer that* $y \leftarrow b$*.*

Example 2. Suppose that a propagator $x \neq y$ is informed that $x \leftarrow a$, hence it determines that y cannot also be assigned to a. The propagator will carry out pruning $y \not\leftarrow a$. The *explanation* for $y \not\leftarrow a$ is $\{x \leftarrow a\}$, intuitively because the latter set is sufficient for the propagator to carry out the former pruning, i.e., $y \not\leftarrow a$ *because* $x \leftarrow a$.

Explanations 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. To ensure that all (dis-)assignments are labelled correctly, the solver will also generate explanations for cases where (i) a variable is set because only one value remains and (ii) a value is pruned because the variable has been assigned to a different value.

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* (i.e., how many decision assignments made so far?), and s is a *sequence number* within the decision depth (i.e., i for the i^{th} (dis-)assignment, starting at 0). For example, 2.0 for the decision assignment at depth 2 and 0.7 for the eighth (dis-)assignment at the root node (depth 0).

The final difference between learning and the standard solver is the way that conflicts are handled. Rather than backtracking, a conflict analysis procedure will run and this is when the explanations are exploited. The aim is to obtain a new constraint that is added to the constraint store after backtracking, to prevent similar conflicts occuring again. The analysis procedure used in g-learning is quite similar to that used in conflict clause learning SAT solvers (e.g. [32]). That is, starting with a *clause* (i.e., disjunction of (dis-)assignments), selected (dis-)assignments are replaced by their explanation until a suitable new clause is derived. Finally the new clause is posted into the solver and search continues. Search now avoids entering certain unnecessary branches of search because the new clause boosts inference.

A more detailed discussion of the g-learning algorithm is, unfortunately, beyond the scope of this paper, however it is important to emphasise certain essential properties of the explanations used to label (dis-)assignments (see [23] for equivalent properties used in SMT solvers). Suppose explanation $\{d_1, \ldots, d_k\}$ labels pruning $v \nleftarrow a$:

Property 1. At least one of $d_1, \ldots, d_k$ must have become true at the same decision depth as $v \nleftarrow a$ occurred.

Remark 1. Intuitively it means that once the (dis-)assignments in the explanation become true, the pruning must be carried out at that decision depth. The property is true of GAC propagators, for example, but not bounds consistency Z propagators [3]. In a proof of correctness of g-learning it ensures that a firstUIP [32] cut exists.

Property 2. None of $d_1, \ldots, d_k$ may have a depth greater than or equal to $v \nleftarrow a$.

Remark 2. Ensures that causes must precede effects[1].

Now we proceed to describe our new work: Section 3 introduces a way of working out explanations for propagations lazily when they are needed, instead of eagerly as the propagations are done. In Section 4 we show how to specialise this for specific propagators. Finally in Section 5, we show empirically that laziness saves time and space because many explanations stored eagerly are never used.

3 Lazy explanations

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 explanation to be reconstructed efficiently later in the *same branch of the search tree*, this we call *laziness.*

Specifically, when a propagator carries out a pruning (or assignment), it must provide a data record, along with a pointer to a function that takes such an object as a parameter[2]. The record and function are stored by the runtime system for later retrieval. Later in search, when an explanation is requested by conflict analysis (or some other procedure), the function will be invoked on the record, and it must return a valid explanation for the earlier pruning. In a g-learning

[1] avoiding cycles in the g-learning implication graph[20, 15]

[2] alternatively, objects with a polymorphic method could be used

framework, Properties 1 and 2 must also be satisfied to ensure correctness. It is likely that the function will access propagator state and variable domain state to carry out this task, and it may perform arbitrary computation. Contrast this with an explanation recorded eagerly: the propagator will calculate the explanation at the time of pruning and it will be stored; later on it will be fetched from storage.

In Section 4 we will describe lazy explanation functions for various useful constraints. Of course, explanations can be still done eagerly with no loss of efficiency when it is hard or inconvenient to work out explanations retrospectively for a particular constraint.

The ability to create explanations lazily is only intended to be available later in the *same branch*, while the (dis-)assignment is still valid. This is because domain information for earlier states in the same branch can be reconstructed, and some lazy explainers described in Section 4 will make use of this information.

Since constraint solvers spend most of their time propagating, an overhead at propagation-time is very damaging to the solver as a whole. This is the reason why computing and storing the explanation lazily is attractive. Hence, we seek to store the minimum amount of data that will suffice to calculate the explanation efficiently later.

We now describe the execution of a solver implementing lazy explanations:

Example 3. Suppose that our CSP consists of variables v, w, x, y and z, each with domain $\{1, \ldots, 5\}$; and the set of constraints includes alldiff(v, w, x, y, z), meaning that all the variables must take different values. Suppose that the domains of variables v and w are reduced to $\{1, 2\}$, then the alldiff is able to propagate: v and w have the possibility of values 1 and 2 between them, and since each needs a distinct value both are required. Hence 1 and 2 should be removed from the domains of x, y and z. An eager solver will compute and store the explanation $\{v \neq 3, v \neq 4, v \neq 5, w \neq 3, w \neq 4, w \neq 5\}$ for each pruning. A lazy solver will instead store only a function pointer and a small object containing a pointer to the alldiff propagator; in this way the effort of producing an explanation is delayed and may never need to happen. Suppose that, later in search, the pruning $x \neq 1$ is involved in a domain wipeout. The conflict analysis procedure may request an explanation for the pruning. At this stage an eager solver will fetch it from storage. A lazy solver will instead invoke the stored function on the small object stored earlier, which will execute code to retrospectively compute an explanation (this procedure is described in Section 4.4). The clause the conflict analysis procedure produces, however the explanations are derived, can now be posted into the solver.

Lazy explanations are intended to reduce the overhead that learning places on propagators. To our knowledge, this is the first application of lazy explanations to a CP solver. As we shall say in more detail in Section 6, at least one SAT modulo theories (SMT) solver uses a similar technique, whereby inference engines for specialised theories such as integer linear arithmetic guarantee to provide an explanation for an inference lazily when it is requested. Also a similar technique has been used before in a solver for jobshop problems [31]. Previously techniques like g-learning and CBJ [25] required potentially a lot of data to be collected during search, however now in many cases we need only store two pointers. This

brings CP solvers more in line with SAT solvers, which need only store a single pointer per propagation to enable learning [20].

4 Lazy explanations for constraint propagators

In this section we describe how specific constraint propagators can be made to produce lazy explanations, specifically, what they need to store at propagation-time and what they need to do later if and when the explanation is requested. We include propagators for clauses, less than, table/extensional and alldiff constraints.

Between them, these propagators range from the simplest (clause) to among the most complex (alldiff). This sample of the available constraints serves to expose the core ideas needed to integrate lazy learning into other propagators.

Finally we describe a generic procedure that will work for an arbitrary constraint, to prove that a propagator can always be lazy, whatever constraint it implements.

4.1 Explanations for clauses

If clause $a \leftarrow 1 \vee b \leftarrow 2 \vee c \leftarrow 3 \vee d \leftarrow 4$ causes assignment $d \leftarrow 4$, in order to calculate an explanation later it is sufficient to note only which constraint did it, i.e., to store a pointer to the clause. Before explaining why, we need to define *unit propagation* which is the consistency level used to propagate clauses:

Definition 2. *When all but one (dis-)assignment e_i in a clause $e_1 \vee e_2 \vee \ldots \vee e_r$ are false,* unit propagation *will set e_i to be true.*

Example 4. Suppose that $a \not\leftarrow 1$, $b \not\leftarrow 2$ and $c \not\leftarrow 3$, then the propagator for the clause $a \leftarrow 1 \vee b \leftarrow 2 \vee c \leftarrow 3 \vee d \leftarrow 4$ will set $d \leftarrow 4$, as the remaining disjuncts are all false. This is necessary because at least one disjunct must be true to satisfy the clause.

Now suppose later we are asked to generate an explanation lazily: we know that the pruning was by unit propagation and can use this fact to infer that all of $a \leftarrow 1, b \leftarrow 2$ and $c \leftarrow 3$ were false at that point. Hence the explanation is $\{a \leftarrow 1, b \leftarrow 2, c \leftarrow 3\}$ or informally the negative of the clause itself with $d \leftarrow 4$ removed.

This form of lazy learning is very familiar because it is what SAT solvers do [20]. 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 $v_1 < v_2$ causes pruning $v_1 \not\leftarrow a$; it is sufficient to store only a pointer to the constraint $v_1 < v_2$ to later reconstruct the explanation. a is pruned if and only if all values in v_2 greater than a are removed, since these are the potential supports for a. Hence explanation $\{v_2 \not\leftarrow a+1, \ldots, v_2 \not\leftarrow \max(d_2)\}$ can be computed when required.

In the next example it will be necessary to reconstruct the domain state at the time when the pruning was made, and as we will show these operations can be implemented in $O(1)$ time with the aid of the stored (dis-)assignment depths.

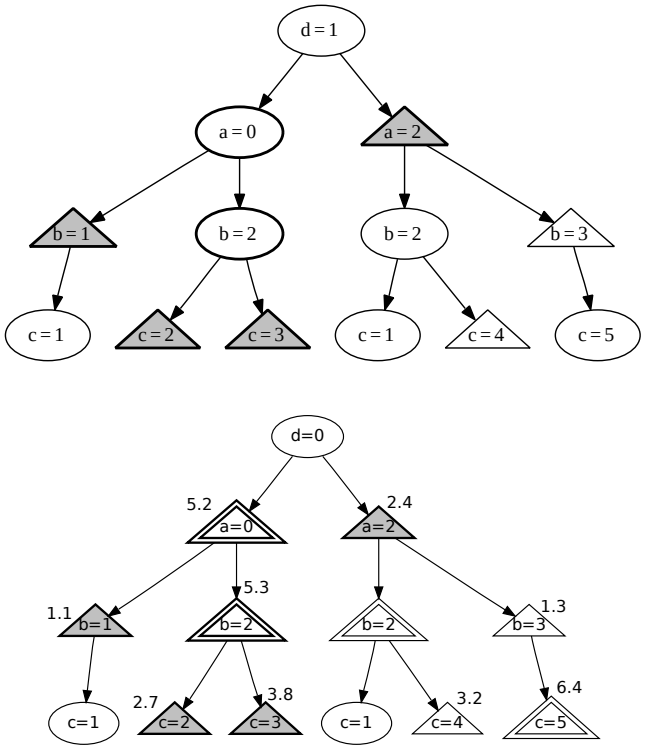

Figure 1. (top) Trie with pruned values shown as triangles, greyed nodes are those included in the explanation and nodes visited in the traversal are bold. (bottom) Same trie but values pruned between the original pruning (at depth 3.9) and the explanation being produced are in double triangles. Pruning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

4.3 Explanations for table

The extensional or "table" constraint is an important part of a constraint library. The user lists the allowed tuples. Hence it can mimic any other constraint, or be used to express an arbitrary relation in a straightforward way where in many cases it would be awkward to express otherwise. For example the relation "married to", $\{(tom, sally), (bob, marie), (sean, jenny)\}$.

Assume we are using an implementation of table [7] where tuples are stored as an array of tries, one per variable, so that all tuples involving a particular variable and value (varval) are readily accessible, as illustrated in Figure 1. For example, the trie at the top of Figure 1 represents every tuple involving $d = 1$, meaning that the set of tuples is $\{(d = 1, a = 0, b = 1, c = 1), (d = 1, a = 0, b = 2, c = 2), (d = 1, a = 0, b = 2, c = 3), (d = 1, a = 2, b = 2, c = 1), (d = 1, a = 2, b = 2, c = 4), (d = 1, a = 2, b = 3, c = 5)\}$.

We say that a varval $x = a$ is pruned when $x \nleftarrow a$. We say that a tuple is *valid* when none of its component varvals are pruned. The propagator works by

ensuring that each varval $v_i = a$ s.t. $a \in d_i$ has at least one support, i.e., there exists at least one valid tuple containing $v_i = a$. If any component of the support is pruned either a new support can be found in the trie, or the $v_i = a$ is pruned.

Such a constraint will prune the varval $v_i = a$ if and only if every tuple containing $v_i = a$ has at least one component varval pruned. We will say that a pruning $v_i \nleftarrow a$ is a *cover* for tuple t iff $v_i = a$ is a component of t. Hence the explanation for $v_i \nleftarrow a$ is a set containing *at least one* cover for each tuple containing $v_i = a$. We 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 traversal of the trie but whenever a node corresponding to a pruned varval is visited add the corresponding pruning to the set and don't recurse any further. Each pruning covers all the tuples beneath the point in the trie when it was added. Figure 1 (top) illustrates this process: when an explanation for $d \leftarrow 1$ is required, the traversal produces $\{b \nleftarrow 1, c \nleftarrow 2, c \nleftarrow 3, a \nleftarrow 2\}$. Note that $b \nleftarrow 3$ and $b \nleftarrow 4$ are not included in the traversal because all supports are covered without them.

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 made *at that time*; i.e., to explain a pruning at depth $a.b$, we would consider only nodes for varvals pruned at a depth less than $a.b$. Such a situation is illustrated in Figure 1 (bottom) where the double lined triangular nodes are not used.

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

The previous examples illustrate that the time and space complexity of lazy explanation generation can match eager evaluation in the worst case, 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.

4.4 Explanations for alldifferent

The alldifferent (alldiff) 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_i to value a whenever $a \in d_i$.

In the following, let r denote the size of an alldiff's scope and d the size of the largest domain. Régin's GAC alldiff propagation algorithm [27] 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

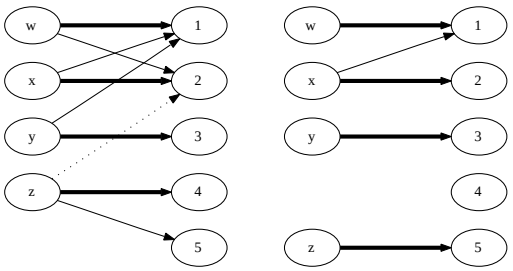

Figure 2. (left)Variable value graph at time of original pruning (right) Same graph at time of explanation

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:

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 earlier consumed the pruned value, 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 variables in 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 of 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 explanations for

an arbitrary propagator with unknown implementation: the explanation of a (dis-)assignment is just the set of all 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 explanation can always be produced lazily, although by specialising for each propagator as described above we will obtain smaller explanations 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 the minion solver[3], a highly optimised solver that didn't originally contain any learning or explanation mechanisms [6]. We make implementation decisions so that compared to the experiments in [15] we are varying only the method used to produce explanations. 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.

To produce explanations we store an small object with a polymorphic function that produces an explanation. For eager, the stored explanation is returned immediately; for lazy, the function implements an algorithm to calculate the explanation. Explanations are not memoized, hence they may be calculated multiple times. This decision has no effect on the eager implementation, although it may be to the detriment of the lazy implementation. Learned clauses are propagated by the 2-watch literal scheme [21].

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.

With the exception of antichain, these were all produced by Tailor [9] using instances from the CSPXML repository [18] 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.

[3] specifically revision number 1885

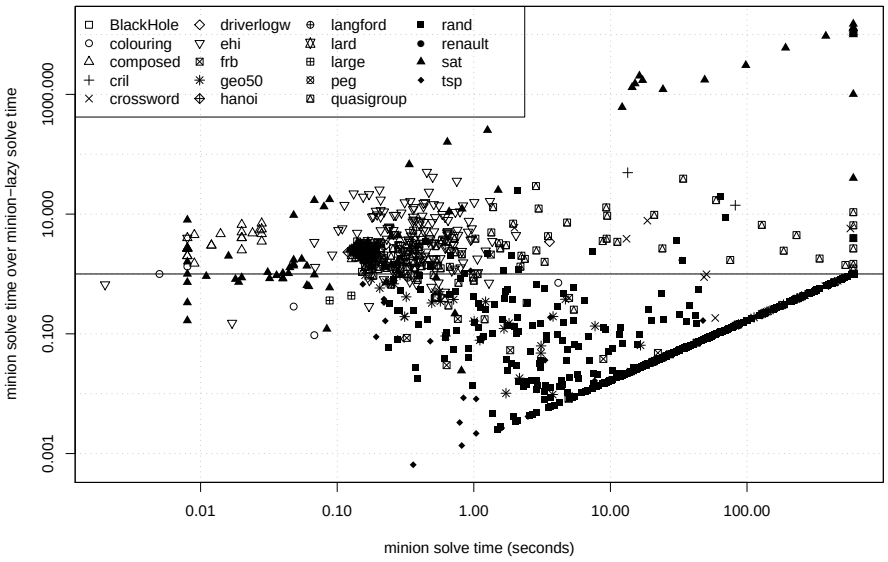

Figure 3. Scatterplot showing runtime comparison for minion versus minion-lazy. Each point is a result for a single instance. The x-axis is the solve time for minion (i.e., excluding set up time which is identical for both). The y-axis gives the speedup from using minion-lazy instead of minion. A ratio 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.

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 10000x.

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. For example a point with y-axis 20 needed just 1/20th of the explanations.

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. This is an particularly significant because the solver spends only part of its time computing explanations. In fact, on some instances we approach the maximum possible speedup, i.e., time to generate explanations approaches 0. 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) and found that the difference between lazy and eager is statistically significant at the 1% level.

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 Related work

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 for CSP solvers. Some of 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 failing partial assignment. Assignments are removed selectively while maintaining the property that it cannot be extended to a solution. Rather than using explanations to do this, a precomputed table is used to establish if a value is compatible with all values in another variable. The jump-back scheme piggy-backs on conflict-directed backjumping (CBJ) [25, 26]. 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. [29] describe how to build and use generic explanations that are not made by the propagator. Jussien et al. [28, 12, 13] described how to produce explanations (consisting of assignments only) for global constraints for various purposes including integrating

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

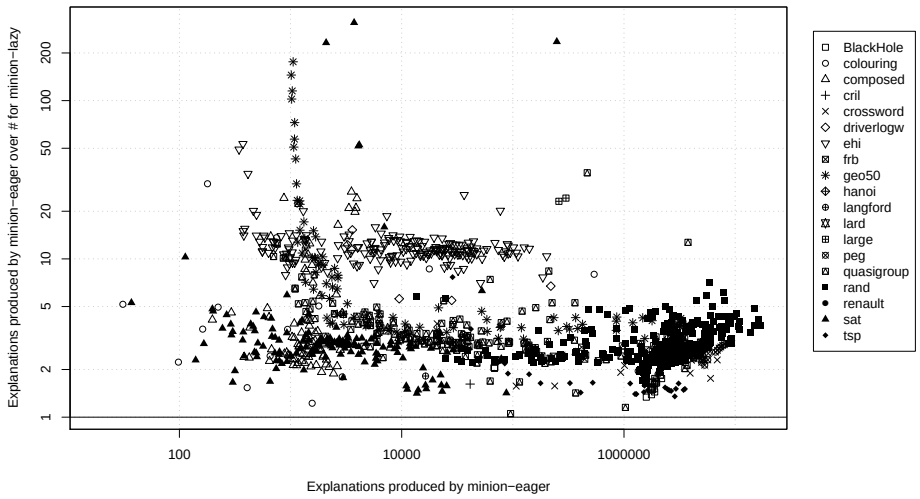

Figure 4. Scatterplot showing comparison of number of explanations produced by minion-lazy versus minion-eager, fewer for instances above the line.

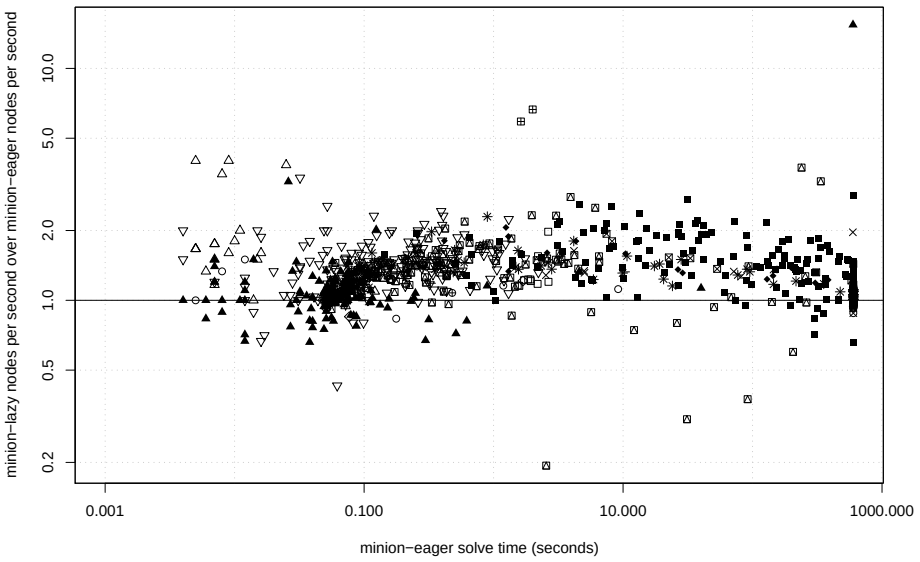

Figure 5. Scatterplot comparing nodes per second for minion-lazy versus minion-eager, more for instances above the line. For legend see Figure 4.

MAC and dynamic backtracking, user interaction and learning constraints. They were produced eagerly by propagators. Cambazard et al. [2] used eagerly built explanations for variable and value ordering heuristics. G-learning (see Section 2.2) is a significant improvement on previous learning schemes as it makes the insight that g-explanations, i.e., explanations containing dis-assignments as well as assignments, are far superior in terms of compactness and propagation power when combined to create new constraints. As described above it too uses eager explanations. Lazy clause generation [24] makes use of explanations in order to derive so-called propagation rules which are then posted into a SAT solver. Explanations are derived eagerly; indeed it would not make sense to derive them lazily as they are always propagated immediately. Explanations have also been computed eagerly for constraints implemented as BDDs [30].

Hence to the best of our knowledge the idea of lazy explanations are unexploited in CSP algorithms, although they have potential in many areas. We now summarise similar ideas from related fields.

Satisfiability modulo theories (SMT) solvers use a form of lazy explanations [23], whereby theories are able to retrospectively produce explanations for the assignments they make to SAT variables. The motivation for this technique is the same as our motivation: to reduce the number of unnecessary explanations produced. [23] gives empirical results proving the effectiveness of the technique in SMT solvers. We have shown that the technique is also valid and effective in CSP solvers. Currently, the SMT community are incorporating constraint propagation algorithms into their tools (see SAT 09 invited talk [22]). Hence this paper also contributes to SMT by describing how to produce lazy explanations for several key global constraints that are currently being integrated into SMT solvers.

In [31] explanations (called justifications) are computed lazily in a specialised solver for the jobshop problem involving only specialised scheduling constraints. They are used to implement conflict directed backjumping. Empirical results show that between 25 and 80% of explanations are never needed, but the paper does not empirically justify that time is saved by their use.

7 Conclusions and Future Work

We have introduced *lazy explanations* for constraint propagation, in which explanations 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 explanations 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.

Acknowledgements Thanks to George Katsirelos for invaluable discussions about g-learning. Thanks to Lars Kotthoff, Pete Nightingale and Andrea Rendl for helping with the preparation of the paper and experiments. Thanks to Chris Jefferson for his assistance with algorithms. Thanks to the anonymous referees for their help in improving the paper. The authors are supported by ESPRC grant number EP/E030394/1 - "Watched Literals and Learning for Constraint Programming".

Bibliography

[1] F. Boussemart, F. Hemery, C. Lecoutre, and L. Sais. Boosting systematic search by weighting constraints. In *ECAI 04*, pages 482–486, August 2004.

[2] Hadrien Cambazard and Narendra Jussien. Identifying and exploiting problem structures using explanation-based constraint programming. In *CP-AI-OR 05*, volume 3524 of *LNCS*, pages 94–109.

[3] 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.

[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 *AAAI-94*, volume 1, pages 294–300. AAAI Press, 1994.

[6] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *ECAI*, pages 98–102, 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. *AIJ*, 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. *JAIR*, 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 *CP*, number 1894 in LNCS, pages 249–261, September 2000.

[14] George Katsirelos, December 2008. Personal correspondence.

[15] George Katsirelos. *Nogood Processing in CSPs*. PhD thesis, University of Toronto, Jan 2009. `http://hdl.handle.net/1807/16737`.

[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] Christophe Lecoutre. Cspxml benchmark repository. `http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html`.

[19] Alan K. Mackworth. On reading sketch maps. In *IJCAI*, pages 598–606, 1977.

[20] 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, November 1996.

[21] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *DAC 01*, 2001.

[22] Robert Nieuwenhuis. Sat modulo theories: Enhancing sat with special-purpose algorithms. In *SAT*, page 1, 2009.

[23] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.

[24] Olga Ohrimenko, Peter Stuckey, and Michael Codish. Propagation = lazy clause generation. In C. Bessière, editor, *CP 07*, volume 4741 of *LNCS*, pages 544–558.

[25] Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Computational Intelligence, Volume 9, Number 3*, pages 268–299, 1993.

[26] 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.

[27] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, pages 362–367, 1994.

[28] 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.

[29] 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.

[30] Sathia Moorthy Subbarayan. Efficient reasoning for nogoods in constraint solvers with BDDs. In *PADL 2008*, volume 4902 of *LNCS*, pages 53–67.

[31] Petr Vilím. Computing explanations for the unary resource constraint. In *CPAIOR*, volume 3524 of *LNCS*, pages 396–409. Springer, 2005.

[32] 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.