Inter-instance Nogood Learning in Constraint Programming

Geoffrey Chu and Peter J. Stuckey

National ICT Australia, Victoria Laboratory,
Department of Computer Science and Software Engineering,
University of Melbourne, Australia
{gchu,pjs}@csse.unimelb.edu.au

Abstract. Lazy Clause Generation is a powerful approach to reducing search in Constraint Programming. This is achieved by recording sets of domain restrictions that previously led to failure as new clausal propagators called nogoods. This dramatically reduces the search and provides orders of magnitude speedups on a wide range of problems. Current implementations of Lazy Clause Generation only allows solvers to learn and utilize nogoods *within* an individual problem. This means that everything the solver learns will be forgotten as soon as the current problem is finished. In this paper, we show how Lazy Clause Generation can be extended so that nogoods learned from one problem can be retained and used to significantly speed up the solution of other, similar problems.

1 Introduction

Lazy Clause Generation (LCG) [8, 5] is a powerful approach to reducing search in Constraint Programming (CP). Finite domain propagation is instrumented to record an explanation for each inference. This creates an implication graph like that built by a SAT solver [7], which may be used to derive nogoods that explain the reason for the failure. These nogoods can be propagated efficiently using SAT unit propagation technology, and can lead to exponential reductions in search space on many problems. Lazy clause generation provides state of the art solutions to a number of combinatorial optimization problems such as resource constrained project scheduling [9] and carpet cutting [10].

Current implementations of Lazy Clause Generation derive nogoods which are only valid within the problem in which they were derived. This means the nogoods cannot be correctly applied to other problems and everything that is learned has to be thrown away after the problem is finished. There are many real life situations where a user might want to solve a series of similar problems, e.g., when problem parameters such as customer demands, tasks, costs or availability of resources change. Clearly, it would be beneficial if the nogoods learned in one problem can be used to speedup the solution of other similar problems.

There are many methods that attempt to reuse information learned from solving previous problems in subsequent problems. However, the information learned by such methods differ significantly from those discussed in this paper. Portfolio based methods (e.g., [11]) use machine learning or similar techniques

to try to learn which solver among a portfolio of solvers will perform best on a new problem given characteristics such as the problem size or the properties of its constraint graph. Methods such as [6] attempt to learn effective search heuristics for specific problem domains by using reinforcement learning or similar techniques. In the case where we wish to find the solution to a modified problem which is as similar to the old solution as possible, methods such as [1], which keep track of the value of each variable in the old solution and reuses it as a search heuristic, can be effective. In the special case where we have a series of *satisfaction* problems where each problem is strictly more constrained than the previous one, e.g. satisfaction based optimization, all nogoods can trivially be carried on and reused in the subsequent problems. In this paper, we are interested in the more general case where subsequent problems can be more constrained, less constrained, or simply different because the parameters in some constraints have changed. We show how to generalize Lazy Clause Generation to produce *parameterized nogoods* which can be carried from instance to instance and used for an entire problem class.

2 Background

Let $\equiv$ denote syntactic identity, $\Rightarrow$ denote logical implication and $\Leftrightarrow$ denote logical equivalence. A *constraint optimization problem* is a tuple $P \equiv (V, D, C, f)$, where V is a set of variables, D is a set of domain constraints $v \in D_v, v \in V$, C is a set of constraints, and f is an objective function to minimize (we can write $-f$ to maximize). An assignment θ assigns each $v \in V$ to an element $\theta(v) \in D_v$. It is a *solution* if it satisfies all constraints in C. An assignment θ is an *optimal solution* if for all solutions θ', $\theta(f) \leq \theta'(f)$. In an abuse of notation, if a symbol C refers to a set of constraints $\{c_1, \ldots, c_n\}$, we will often also use the symbol C to refer to the conjunction $c_1 \wedge \ldots \wedge c_n$.

CP solvers solve CSP's by interleaving search with inference. We begin with the original problem at the root of the search tree. At each node in the search tree, we propagate the constraints to try to infer variable/value pairs which cannot be taken in any solution to the problem. Such pairs are removed from the current domain. If some variable's domain becomes empty, then the subproblem has no solution and the solver backtracks. If all the variables are assigned and no constraint is violated, then a solution has been found and the solver can terminate. If inference is unable to detect either of the above two cases, the solver further divides the problem into a number of more constrained subproblems and searches each of those in turn.

In an LCG solver, each propagator is instrumented to explain each of its inferences with a clause called the *explanation.* Each clause consists of *literals* of the form $x = v, x \neq v, x \geq v$ or $x \leq v$ where x is a variable and v is a value.

Definition 1. *Given current domain D, suppose the propagator for constraint c makes an unary inference m, i.e., $c \wedge D \Rightarrow m$. An* explanation *for this inference is a clause: $expl(m) \equiv l_1 \wedge \ldots \wedge l_k \rightarrow m$ s.t. $c \Rightarrow expl(m)$ and $D \Rightarrow l_1 \wedge \ldots \wedge l_k$.*

The explanation $expl(m)$ explains why m has to hold given c and the current domain D. We can consider $expl(m)$ as the fragment of the constraint c from which we inferred that m has to hold. For example, given constraint $x \leq y$ and

current domain $x \in \{3, 4, 5\}$, the propagator may infer that $y \geq 3$, with the explanation $x \geq 3 \rightarrow y \geq 3$.

These explanations form an acyclic implication graph. Whenever a conflict is found by an LCG solver, the implication graph can be analyzed in order to derive a set of sufficient conditions for the conflict. This is done by repeatedly resolving the conflicting clause (the clause explaining the conflict) with explanation clauses, resulting in a new nogood for the problem.

Example 1. Let $C_1 \equiv \{x_2 < x_3, x_1 + 2x_2 + 3x_3 \leq 13\}$. Suppose we tried $x_1 = 1$ and $x_2 = 2$. We would infer $x_3 \geq 3$ from the first constraint with explanation: $x_2 \geq 2 \rightarrow x_3 \geq 3$. The second constraint would then fail with explanation: $x_1 \geq 1 \wedge x_2 \geq 2 \wedge x_3 \geq 3 \rightarrow \mathit{false}$. Resolving the conflicting clause with the explanation clause gives the nogood: $x_1 \geq 1 \wedge x_2 \geq 2 \rightarrow \mathit{false}$.

Let $\mathit{expls}(n)$ be the set of explanations from which a nogood n was derived. Then $\mathit{expls}(n) \Rightarrow n$.

3 Parameterized Nogoods

Suppose we want to solve several similar problem instances from the same problem class. Currently, Lazy Clause Generation produces nogoods which are only correct within the instance in which it was derived, thus we cannot carry such nogoods from one instance to the next and reuse them. In this section, we show how we can generalize the nogoods produced by Lazy Clause Generation to *parameterized* nogoods which are valid for a whole problem class.

Each nogood derived by Lazy Clause Generation represents a resolution proof that a certain subtree in the problem contains no solutions. For us to correctly reuse this nogood in a different problem, we have to show that the resolution proof is valid in the other problem. We have the following result:

Theorem 1. *Let $P_1 \equiv (V, D, C_1, f)$ and $P_2 \equiv (V, D, C_2, f)$ be two constraint optimization problems. Let n be a nogood derived while solving P_1. If $C_2 \Rightarrow \mathit{expls}(n)$, then n is also a valid nogood for P_2.*

Proof. $C_2 \Rightarrow \mathit{expls}(n) \Rightarrow n$.

Theorem 1 tells us that if every explanation used to derive a nogood n is also implied by a second problem P_2, then n is also a valid nogood in P_2.

Example 2. Let $C_1 \equiv \{x_1 < x_2, x_1 + 2x_2 \leq 9\}$ and $C_2 \equiv \{x_1 + 1 < x_2, x_1 + 2x_2 \leq 10\}$. Suppose we tried $x_1 = 3$ in the first problem. We would infer $x_1 \geq 3 \rightarrow x_2 \geq 4$ from the first constraint, and the second constraint would then fail with $x_1 \geq 3 \wedge x_2 \geq 4 \rightarrow \mathit{false}$. The nogood would simply be $x_1 \geq 3 \rightarrow \mathit{false}$. Now, this nogood is also valid in the second problem because: $x_1 + 1 < x_2 \Rightarrow x_1 \geq 3 \rightarrow x_2 \geq 4$ and $x_1 + 2x_2 \leq 10 \Rightarrow x_1 \geq 3 \wedge x_2 \geq 4 \rightarrow \mathit{false}$, so both explanations used to derive the nogood are implied in the second problem.

Note that the constraints in the second problem do not have to be the same as the first. They can be stronger (like the first constraint in Example 2) or weaker (like the second constraint in Example 2), and there can be more or

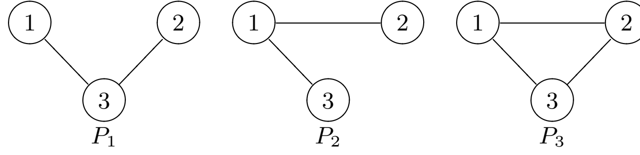

Fig. 1. 3 different graphs for colouring

fewer constraints, as long as the constraints are strong enough to imply all the explanations used to derive the nogood in the first problem.

To determine whether a nogood can be reused in a different problem, we need an efficient way to keep track of whether all the explanations used to derive a nogood are implied in the new problem. We can alter the problem models in order to achieve this. Instead of modeling each instance as an individual constraint problem, we create a generic *problem class model* which is then parameterized to produce the individual instances. That is, we create a problem class model $P_{class} \equiv (V \cup Q, D, C, f)$, where Q are *parameter variables*. Individual instances P_i are then created by fixing the variables in Q to instance specific values R_i.

Example 3. Consider the graph coloring problem with n nodes. Let $Q \equiv \{a_{i,j} \mid i, j = 1, \ldots, n\}$ be a set of Boolean parameter variables representing whether there is an edge between node i and j. We can define $V \equiv \{v_1, \ldots, v_n\}$, $D \equiv \{v_i \in \{1, \ldots, n\}$, $C \equiv \{a_{i,j} \rightarrow v_i \neq v_j\}$ and $f = max(v_i)$. Each instance would then be created by setting the variables in Q to certain values to represent the adjacency matrix for that instance.

In the traditional way of modeling, parameters are considered as constants and we have separate problems for each problem instance. In our approach however, parameters are variables and there is only a single problem for the entire problem class. Since the parameters are now variables instead of constants, when LCG is used on an instance, the explanations generated by the propagators will include literals on the parameter variables. These additional literals describe sufficient conditions on the parameter values to make the inferences valid. When such parameterized explanations are resolved together to form a nogood, the nogood will also have literals describing sufficient conditions on the parameter values to make the nogood valid. Thus each nogood becomes a *parameterized nogood* that is valid across the whole problem class. On instances where all the conditions on the parameters are satisfied, the nogood will be *active* and will be able to prune things as per normal. On instances where any of the conditions on the parameters are not satisfied, the nogood will be *inactive* (trivially satisfied) and will not prune anything.

Example 4. Consider three graph coloring instances: P_1, where $a_{1,2} = \mathit{false}, a_{1,3} = \mathit{true}, a_{2,3} = \mathit{true}$, P_2, where $a_{1,2} = \mathit{true}, a_{1,3} = \mathit{true}, a_{2,3} = \mathit{false}$, and P_3, where $a_{1,2} = \mathit{true}, a_{1,3} = \mathit{true}, a_{2,3} = \mathit{true}$, illustrated in Figure 1. Suppose in P_1, we are looking for solutions with $f \leq 2$ and we made search decisions $v_1 = 1, v_2 = 2$. In the unparameterized model, we would use explanations: $f \leq 2 \rightarrow v_3 \leq 2$, $v_1 = 1 \rightarrow v_3 \neq 1$, $v_2 = 2 \rightarrow v_3 \neq 2$, $v_3 \neq 1 \wedge v_3 \neq 2 \wedge v_3 \leq 2 \rightarrow \mathit{false}$

and would derive a nogood $v_1 = 1 \wedge v_2 = 2 \wedge f \leq 2 \rightarrow \mathit{false}$. Now, it is incorrect to apply this nogood in P_2, because it is simply not true. For example $v_1 = 1, v_2 = 2, v_3 = 2$ is a perfectly valid solution for P_2. In the parameterized model, we would use the explanations: $f \leq 2 \rightarrow v_3 \leq 2$, $v_1 = 1 \wedge a_{1,3} \rightarrow v_3 \neq 1$, $v_2 = 2 \wedge a_{2,3} \rightarrow v_3 \neq 2$, $v_3 \neq 1 \wedge v_3 \neq 2 \wedge v_3 \leq 2 \rightarrow \mathit{false}$ and would derive a nogood $a_{1,3} \wedge a_{2,3} \wedge v_1 = 1 \wedge v_2 = 2 \wedge f \leq 2 \rightarrow \mathit{false}$, which correctly encapsulates the fact that the nogood is only valid if the graph has edges between node 1 and 3 and node 2 and 3. The parameterized nogood can be correctly applied to any instance of the graph coloring problem. It is inactive in P_2 because $a_{2,3} = \mathit{false}$ in P_2, but it might prune something in P_3 because $a_{1,3} = a_{2,3} = \mathit{true}$ in P_3.

4 Implementation

Naively, we could implement the parameter variables as actual variables with constraints over them. Then simply running the normal LCG solver on this model will generate parameterized nogoods. However, such an implementation is less efficient than using an instance specific model where the parameters are considered as constants. For example, a linear constraint: $\sum a_i x_i$ where a_i are parameters would be a simple linear constraint in an instance model, but would be a quadratic constraint if we consider parameter variables as actual variables. We can improve the implementation by taking advantage of the fact that the parameters will always be fixed when we are solving an instance. To do this we use normal propagators which treat parameters as constants, but alter their explanations so that they include literals representing sufficient conditions on the parameters to make the inference true.

For example, given a linear constraint $a_1x_1 + a_2x_2 + a_3x_3 \leq 10$, $a_1 = 1, a_2 = 2, a_3 = 3$ and current bounds $x_1 \geq 1$, $x_2 \geq 2$, we can infer $x_3 \leq 1$, and we would explain it using: $a_1 \geq 1 \wedge a_2 \geq 2 \wedge a_3 \geq 3 \wedge x_1 \geq 1 \wedge x_2 \geq 2 \rightarrow x_3 \leq 1$. Given a constraint c which may be added to or removed from an instance depending on a Boolean parameter b, we would modify the explanations for c's inferences by adding the literal b to each explanation. Given a *cumulative* constraint where task durations, resource usage and the capacity of the machine are parameters, we would add lower bound literals on the duration and resource usage of each task involved in the inference, and an upper bound literal on the machine capacity to each explanation. The modifications to the explanations of other parameterized constraints are similarly straightforward and we do not describe them all.

A LCG solver can generate an enormous number of parameterized nogoods during search, most of which are not particularly useful. Clearly, it would be inefficient to retain all of these nogoods. We take advantage of the inbuilt capabilities of LCG solvers for deleting useless nogoods. The LCG solver CHUFFED maintains an activity score for each nogood based on how often it is used. When the number of nogoods in the constraint store reaches 100000, the least active half are deleted. We only reuse the, at most 100000, parameterized nogoods which survive till the end of the solve. At the beginning of each new instance, we check each parameterized nogood to see if the conditions on the parameters are satisfied. If not, we ignore the parameterized nogood, as it cannot prune anything in this particular instance. If the conditions are satisfied, we add it to the constraint store and handle it in the same way as nogoods learned during

search, i.e., we periodically remove inactive ones. This ensures that if the parameterized nogoods learned from previous instances are useless, they will quickly be removed and will no longer produce any overhead.

4.1 Strengthening Explanations for Inter-instance Reuse

An important optimization in LCG is to *strengthen* explanations so that the nogoods derived from it are more reusable. For example, consider a constraint: $x_1 + 2x_2 + 3x_3 \leq 13$, and current domains: $x_1 \geq 4, x_2 \geq 3$. Clearly, we can infer that $x_3 \leq 1$. Naively, we might explain this using the current bounds as: $x_1 \geq 4 \wedge x_2 \geq 3 \rightarrow x_3 \leq 1$. This explanation is valid, but it is not the strongest possible explanation. For example, $x_1 \geq 2 \wedge x_2 \geq 3 \rightarrow x_3 \leq 1$ is also a valid explanation and is strictly stronger logically. Using these stronger explanations result in stronger nogoods which may prune more of the search space. It is often the case that there are multiple ways to strengthen an explanation and it is not clear which one is best. For example, $x_1 \geq 4 \wedge x_2 \geq 2 \rightarrow x_3 \leq 1$ is another way to strengthen the explanation for the above inference.

In the context of inter-instance learning, there is an obvious choice of which strengthening to pick. We can preferentially strengthen the explanations so that they are more reusable across different problems. We do this by weakening the bounds on parameter variables in preference to those on normal variables. For example, if x_1 was a parameter variable and x_2 was a normal variable, we would prefer the first strengthening above rather than the second, as that explanation places weaker conditions on the parameter variables and will allow the nogood to prune things in more instances of the problem class.

4.2 Hiding Parameter Literals

The linear constraint is particularly difficult for our approach as its explanations often involve very specific conditions on the parameters, and these conditions might not be repeated in other instances of the problem we are interested in. For example, suppose: $\sum_{i=1}^{k} a_i x_i \leq m$ where a_i are positive parameter variables and x_i are normal variables. Suppose each a_i is fixed to a value of r_i, and each x_i is fixed to a value of b_i and we have a failure. A naive explanation of this inference would be of the form: $\wedge_{i=1}^{k}(a_i \geq r_i \wedge x_i \geq b_i) \rightarrow \mathit{false}$. This places a lower bound condition on all of the a_i involved in the linear constraint, which may be hard to meet in any other instance of the problem. We can improve the situation by decomposing long linear constraints into ternary linear constraints involving partial sum variables. This serves to "hide" some of the parameters away and makes the explanation more reusable.

Example 5. Suppose we had constraints: $x_3 < x_4$ and $a_1x_1+a_2x_2+a_3x_3+a_4x_4 \leq 29$, and in this instance, the parameters are set to $a_1 = 1, a_2 = 2, a_3 = 3, a_4 = 4$. Suppose we tried $x_1 = 1$, $x_2 = 2$, $x_3 = 3$. We can infer that $x_4 \geq 4$ with explanation $x_3 \geq 3 \rightarrow x_4 \geq 4$. The second constraint then fails with explanation: $(a_1 \geq 1 \wedge a_2 \geq 2 \wedge a_3 \geq 3 \wedge a_4 \geq 4) \wedge x_1 \geq 1 \wedge x_2 \geq 2 \wedge x_3 \geq 3 \wedge x_4 \geq 4 \rightarrow \mathit{false}$, leading to nogood: $(a_1 \geq 1 \wedge a_2 \geq 2 \wedge a_3 \geq 3 \wedge a_4 \geq 4) \wedge x_1 \geq 1 \wedge x_2 \geq 2 \wedge x_3 \geq 3 \rightarrow \mathit{false}$. The condition on the parameters: $(a_1 \geq 1 \wedge a_2 \geq 2 \wedge a_3 \geq 3 \wedge a_4 \geq 4)$ is difficult to satisfy. However, suppose we decomposed the linear constraint into: $a_1x_1 + a_2x_2 \leq s_2, s_2 + a_3x_3 \leq s_3, s_3 + a_4x_4 \leq 29$. Now, after inferring $x_4 \geq 4$

from the first constraint, we would have a chain of inferences and explanations: $(a_1 \geq 1 \wedge a_2 \geq 2) \wedge x_1 \geq 1 \wedge x_2 \geq 2 \rightarrow s_2 \geq 5$, $(a_3 \geq 3) \wedge s_2 \geq 5 \wedge x_3 \geq 3 \rightarrow s_3 \geq 14$, $(a_4 \geq 4) \wedge s_3 \geq 14 \wedge x_4 \geq 4 \rightarrow \mathit{false}$. The nogood would be derived by resolving the last two clause with $x_3 \geq 3 \rightarrow x_4 \geq 4$, which gives the nogood: $(a_3 \geq 3 \wedge a_4 \geq 4) \wedge s_2 \geq 5 \wedge x_3 \geq 3 \rightarrow \mathit{false}$. This nogood only has conditions on a_3 and a_4 and is strictly stronger logically. By introducing the partial sum variables, the conditions on a_1 and a_2 have been "hidden" into the bound literal on s_2 instead, producing a more reusable nogood.

5 Experiments

We evaluate our method on four problems. We briefly describe each problem, their parameters, and situations where we may wish to solve several similar instances of the problem.

Radiation Therapy Problem. In the Radiation Therapy Problem [2], a doctor develops a treatment plan for a patient consisting of an intensity matrix describing the amount of radiation to be delivered to each part of the treatment area. The aim is to find the configuration of collators and beam intensities which minimizes the total amount of radiation delivered and the treatment time. The intensities are parameters. The doctor may alter the treatment plan (change some of the intensities) and we may wish to re-optimize. We use instances of with 15 rows, 12 columns and a max intensity of 10.

Minimization of Open Stacks Problem. In the Minimization of Open Stacks Problem (MOSP) [4], we have a set of customers each of which requires a subset of the products. The products are produced one after another. Each customer has a stack which must be opened from the time when the first product they require is produced till the last product they require is produced. The aim is to find the production order which minimizes the number of stacks which are open at any time. The parameters are whether a customer requires a certain product. Customers may change their orders and we may wish to re-optimize the schedule. We use instances with 35 customers and a shared product density of 0.2.

Graph Coloring. The existence of an edge between each pair of nodes is a parameter. Edges may be added or removed and we may wish to re-optimize. We use instances with 55 nodes and an edge density of 0.33.

Knapsack. In the 0-1 Knapsack Problem, the value, weight and availability of items are parameters. New items might become available, or old ones might become unavailable and we may wish to re-optimize. We use instances with 100 items.

For each of these problems, we generate 100 random instances. From each of these base instances, we generate modified versions of the instance where 1%, 2%, 5%, 10% or 20% of the parameters have been randomly changed. The instances are available online [3]. We solve these instances using the Lazy Clause

Table 1. Comparison of solving (a) Radiation Therapy instances, (b) MOSP instances, (c) Graph Coloring instances, and (d) Knapsack instances: from scratch (scratch) and solving them making use of parameterized nogoods (para) from a similar instance

(a) Radiation Therapy

diff	scratch		para		reuse	speedup
1%	7.10	23000	0.08	32	97%	88.8
2%	7.25	23219	0.31	433	93%	23.8
5%	7.03	22556	1.17	2496	81%	6.02
10%	7.57	23628	2.52	6198	74%	3.00
20%	7.68	23326	4.35	11507	55%	1.77

(b) MOSP

diff	scratch		para		reuse	speedup
1%	38.67	111644	1.03	921	93%	37.5
2%	41.65	114358	4.57	6700	88%	9.11
5%	42.80	111745	31.02	75600	71%	1.38
10%	47.80	95521	40.77	88909	57%	1.17
20%	27.37	86231	33.26	88186	45%	0.82

(c) Graph Coloring

diff	scratch		para		reuse	speedup
1%	15.12	54854	7.72	17026	74%	1.96
2%	18.78	61630	16.17	35994	61%	1.16
5%	23.65	70710	24.96	52026	35%	0.94
10%	45.14	96668	45.54	79763	20%	0.99
20%	48.12	101668	45.96	87431	9%	1.04

(d) Knapsack

diff	scratch		para		reuse	speedup
1%	18.21	48714	6.74	11339	100%	2.70
2%	18.47	48640	7.32	13042	100%	2.52
5%	18.80	49154	16.91	37786	100%	1.11
10%	19.38	49298	21.12	44952	100%	0.92
20%	20.83	50007	24.15	49387	100%	0.86

Generation solver CHUFFED running on 2.8 GHz Xeon Quad Core E5462 processors. As a baseline, we solve every instance from scratch (scratch). To compare with our method, we first solve each base instance and learn parameterized nogoods from it. We then solve the corresponding modified versions while making use of these parameterized nogoods (para). The geometric mean of the run times in seconds and the nodes required to solve each set of 100 instances is shown in Table 1. We also show the geometric mean of the percentage of parameterized nogoods which are active in the second instance of each pair of instances (reuse), and the speedup (speedup).

As can be seen from the results, parameterized nogoods can provide significant reductions in node count and run times on a variety of problems. The speedups vary between problem classes and are dependent on how similar the instance is to one that has been solved before. Dramatic speedups are possible for Radiation and MOSP when the instances are similar enough, whereas the speedups are smaller for Knapsack and Graph Coloring. The more similar an instance is to one that has been solved before, the greater the number of parameterized nogoods which are active in this instance and the greater the speedup tends to be. When the instance is too dissimilar, parameterized nogoods provide little to no benefit.

The percentage of parameterized nogoods which are active in the second instance is highly dependent on the problem class. This is because depending on the structure of the problem, each parameterized nogood can involve a small or large number of the instance parameters. The fewer the parameters involved, the fewer the conditions on the parameters and the more likely it is that the nogood will be reusable in another instance. The “hiding parameter literals” optimization described in Section 4.2 is clearly beneficial for the Knapsack Problem, raising the reusability to 100%. Without it, few of the parameterized nogoods are active in the second instance and there is no speedup (not shown in table).

While parameterized nogoods must be active in order to provide any pruning, there is no guarantee that an active nogood will actually provide "useful" pruning. This can be seen in the results for Knapsack, where despite the fact that all the parameterized nogoods can potentially prune something, they do not prune anything useful when the second instance is too different from the first.

6 Conclusion

We have generalized the concept of nogoods, which are valid only for an instance, to *parameterized nogoods* which are valid for an entire problem class. We have described the modifications to a Lazy Clause Generation solver required to generate such parameterized nogoods. We evaluated the technique experimentally and found that parameterized nogoods can provide significant speedups on a range of problems when several similar instances of the same problem need to be solved. The more similar the instances are, the greater the speedup from using parameterized nogoods.

Acknowledgments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council. This work was partially supported by Asian Office of Aerospace Research and Development (AOARD) Grant FA2386-12-1-4056.

References

1. Ignasi Abío, Morgan Deters, Robert Nieuwenhuis, and Peter J. Stuckey. Reducing chaos in SAT-like search: Finding solutions close to a given one. In *SAT*, pages 273–286, 2011.
2. Davaatseren Baatar, Natashia Boland, Sebastian Brand, and Peter J. Stuckey. Minimum Cardinality Matrix Decomposition into Consecutive-Ones Matrices: CP and IP Approaches. In Pascal Van Hentenryck and Laurence A. Wolsey, editors, *Proceedings of the 4th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 4510 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2007.
3. G. Chu. Interproblem nogood instances. `www.cis.unimelb.edu.au/~pjs/interprob/`.
4. Geoffrey Chu and Peter J. Stuckey. Minimizing the Maximum Number of Open Stacks by Customer Search. In Ian P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2009.
5. Thibaut Feydy and Peter J. Stuckey. Lazy Clause Generation Reengineered. In Ian P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 352–366. Springer, 2009.
6. Pat Langley. Learning effective search heuristics. In *IJCAI*, pages 419–421, 1983.
7. 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*, pages 530–535. ACM, 2001.
8. Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = Lazy Clause Generation. In Christian Bessiere, editor, *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.

9. Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark Wallace. Why Cumulative Decomposition Is Not as Bad as It Sounds. In Ian P. Gent, editor, *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 746–761. Springer, 2009.
10. Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, 2011.
11. Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for sat. *J. Artif. Intell. Res. (JAIR)*, 32:565–606, 2008.