Learning from Learning Solvers

Maxim Shishmarev[1],
Christopher Mears[12], Guido Tack[12], and Maria Garcia de la Banda[12]

[1] Faculty of IT, Monash University, Australia
[2] Data61/CSIRO, Australia
{maxim.shishmarev, chris.mears, guido.tack, maria.garciadelabanda}@monash.edu

Abstract. Modern constraint programming solvers incorporate SAT-style clause learning, where sets of domain restrictions that lead to failure are recorded as new clausal propagators. While this can yield dramatic reductions in search, there are also cases where clause learning does not improve or even hinders performance. Unfortunately, the reasons for these differences in behaviour are not well understood in practice. We aim to cast some light on the practical behaviour of learning solvers by profiling their execution. In particular, we instrument the learning solver *Chuffed* to produce a detailed record of its execution and extend a graphical profiling tool to appropriately display this information. Further, this profiler enables users to measure the impact of the learnt clauses by comparing Chuffed's execution with that of a non-learning solver, and examining the points at which their behaviours diverge. We show that analysing a solver's execution in this way can be useful not only to better understand its behaviour — opening what is typically a black box — but also to infer modifications to the original constraint model that can improve the performance of both learning and non-learning solvers.

1 Introduction

Lazy Clause Generation (LCG) [10, 5] is a powerful solving technique that combines the strengths of Constraint Programming and SAT solving. It works by instrumenting finite domain propagation to record the reasons for each propagation step, thus creating an implication graph like the ones built by a SAT solver [7]. This graph is used to derive nogoods (i.e., reasons for failure) which can be recorded as clausal propagators and propagated efficiently using SAT technology. The combination of constraint propagation and clause learning can dramatically reduce search and greatly improve performance.

Indeed, LCG solvers are the state of the art for solving a number of hard combinatorial optimisation problems, such as Resource Constrained Project Scheduling Problems [12] and the Carpet Cutting Problem [13]. Further, they consistently exhibit better performance than traditional Constraint Programming (CP) solvers for a large number of problems in the annual MiniZinc Challenge [15]. Yet, for some problems, LCG solvers seem to be unable to benefit from the learnt clauses and perform poorly compared to non-learning competitors. The reasons for these differences in behaviour are not well understood in practice, as learning solvers are even more complex than traditional CP solvers. Thus, it is not yet clear to the research community under what circumstances learning is better, or even how to identify when or why a learning solver is performing poorly or not.

The aim of this paper is to cast some light on the practical behaviour of learning solvers by being able to better *profile their execution*. To achieve this, we instrumented the open-source LCG solver Chuffed [4] to provide statistical data regarding the clauses it learnt. We then fed this new data into the profiling tool introduced in [14], which we augmented with additional visualisations to display and analyse the LCG solving process. The long term aim of our research is to identify properties of the search that often indicate good or bad performance. If those properties can be identified, the profiler will be able to automatically focus the user's attention on the parts of the search that show those properties and suggest a reason for the behaviour, considerably simplifying the profiling task. As shown in Section 4, some of the information uncovered by the profiler has significant potential in this regard.

While using our augmented profiling tools on models where Chuffed achieved remarkably good performance, we realised that the clauses learnt by the solver could sometimes be used to modify the model itself, in such a way as to improve its execution for traditional CP solvers. This insight came from profiling clauses whose information was either (a) already expressed in the model by a single constraint, (b) not as strong as one would have expected, or (c) already captured by the model but not in an explicit way. Case (a) hints at a lack of appropriate propagation for a particular constraint in the model. The user might then decide to change the strength of the propagator (if the solver supports this) or modify the constraint to achieve better propagation. Case (b) also hints at a lack of propagation, possibly as part of the interaction between several constraints. The user might then decide to modify the constraints involved or add a new redundant constraint that achieves the desired propagation. Case (c) might suggest new information that could be expressed as a generic redundant constraint and which might increase propagation if added to the model (as it has increased propagation for the learning solver). While adding redundant constraints to a model is a well known method to improve performance, it can also have the opposite effect, depending on whether the redundant constraint helps propagation or not. Inferring useful, new redundant constraints for a given model is extremely difficult, and we are not aware of any proposed system or method capable of doing so. Using learnt clauses to achieve this is therefore an exciting new approach with significant potential. As shown in Section 3, we have already been able to detect clauses that fit in each of the three cases above, and modified the models accordingly obtaining considerable reductions in search effort.

2 Background

Constraint Programming: A finite domain *constraint problem* P is a tuple (C, D, f), where C is a set of constraints, D a *domain* which maps each variable $x \in \mathit{vars}(C)$ to a finite set of integers $D(x)$, and f an objective function (if any). The set C is logically interpreted as the conjunction of its elements, while D is interpreted as $\wedge_{x \in \mathit{vars}(C)} x \in D(x)$. A literal of P is a unary constraint c where $var(c) \in vars(C)$. A CP solver starts from an original problem $P \equiv (C, D)$ and applies propagation to reduce domain D to D' as a fixpoint of all propagators for C. If D' is equivalent to *false*, we say P is failed. If D' fixes all variables, we have found a solution to P. Otherwise, the solver splits P into n subproblems $P_i \equiv (C \wedge c_i, D')$, $1 \leq i \leq n$ where $C \wedge D' \Rightarrow (c_1 \vee c_2 \vee \ldots \vee c_n)$ and where c_i are literals (called *decision literals*), and iteratively searches these.

The search proceeds making decisions until either (1) a solution is found, (2) a failure is detected, or (3) a restart event occurs. In case (1) the search either terminates if

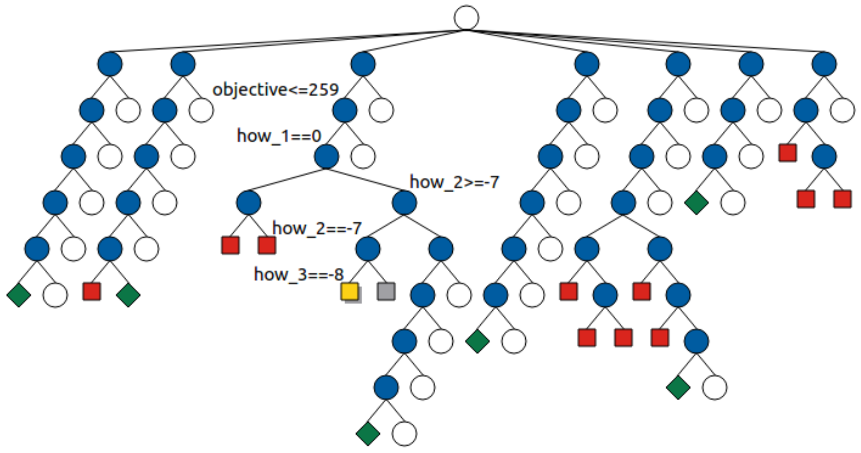

Fig. 1. Search tree for `freepizza` using Chuffed. The path to the highlighted node is labeled.

the model has no objective function, or computes the value of the objective function f, sets a bound for the next value of f to be better (greater or smaller, depending on f) and continues the search for this better value. In case (2), the search usually backtracks to a previous point where a different decision can be made. In case (3) the search starts a new search tree, possibly incorporating new constraints learnt during the previous search.

Profiler: We use the functionality available in the profiler of [14], including its visualisation of the search tree and its tools for convenient navigation and analysis of the search. For example, Figure 1 shows a search tree, where green diamonds denote solutions, red squares (and the highlighted yellow square) failures, grey squares nodes that are skipped due to backjumping, blue circles branching nodes, and white circles either unexplored nodes (in this case skipped due to a restart) or the root of an execution tree with restarts (as is the case in this execution). Labels showing the search decisions can be turned on or off for a given subtree or branch. Of particular interest is the capability to visually *merge* two search trees obtained by, for example, executing the same problem with two different solvers. The result is a combined tree, where the parts where the search is the same are visualised as usual, and those where the searches diverged are depicted as *pentagons* that can be expanded to show the divergent trees. This merging technique is particularly useful in combination with a *replaying* technique, where the search decisions used by a given solver when executing a problem are recorded, and the same decisions are then used to execute the same problem with a different solver. The merged tree then shows exactly where the two solvers behave differently in terms of constraint propagation.

Lazy Clause Generation: LCG solvers [10, 5] can be seen as Satisfiability Modulo Theories solvers [9], where constraint propagators play the role of the theories. They extend CP solvers by instrumenting their propagators to explain the effect of propagation (i.e., domain changes) in terms of literals. In practice, these literals are all either *equality* ($x = d$ for $d \in D(x)$), *disequality* ($x \neq d$) or *inequality* ($x \geq d$ or $x \leq d$) literals. An *explanation* for literal ℓ is $S \rightarrow \ell$, where S is a set of literals. For example, the explanation for the propagator of constraint $x \neq y$ inferring literal $y \neq 5$ given literal

$x = 5$ is $\{x = 5\} \rightarrow y \neq 5$. Explanations make the reasons behind constraint propagation explicit and can be used later when a failure occurs. In LCG solvers, each new literal inferred by a propagator is recorded in a stack in the order it was generated and attached to its explanation. Decision literals are also added to the stack and marked as such. This stack is called the *implication graph*. The *decision level* for any literal is the number of decision literals pushed in the stack before it. Thus, it is similar to the traditional concept of search tree depth in CP.

A nogood N is a set of literals that cannot be extended to a solution. Given an implication graph, LCG solvers compute a nogood by starting with the direct cause of the failure, and then eliminating literals by replacing them with their explanations until only one literal at the current decision level remains. The result is the 1UIP (First Unique Implication Point, [6]) nogood, and its negation ($\neg N$) is added as a clausal propagator. The search then backtracks to the decision level of the second latest literal in the nogood, where it applies the newly learnt clause. Importantly, if the second latest literal is not from the immediately preceding decision level, the search performs a *backjump*, skipping decisions that were unrelated to the failure.

Example 1. Consider the free pizza problem, where customers can get pizzas either by paying for them or by using vouchers. Each voucher is represented by a pair of numbers a/b, indicating the voucher allows customers to get b number of pizzas for free, as long as they pay for a number of pizzas. In addition, none of the b pizzas can be more expensive than the a pizzas. Given a customer who has m such vouchers and wants n pizzas, the aim is to minimise the total price paid for the n pizzas. The model used in the annual MiniZinc Challenge (denoted as `freepizza`) is as follows:

```
1   int: n;     set of int: PIZZA = 1..n;     % number of pizzas wanted
2   array[PIZZA]   of int: price;             % price of each pizza
3   int: m;     set of int: VOUCHER = 1..m;   % number of vouchers
4   array[VOUCHER] of int: buy;               % buy this many to use voucher
5   array[VOUCHER] of int: free;              % get this many free
6
7   set of int: ASSIGN = -m .. m; % i -i 0 (pizza free/paid with voucher i or not)
8   array[PIZZA] of var ASSIGN: how;
9   array[VOUCHER] of var bool: used;
10
11  constraint forall(v in VOUCHER)(used[v]<->sum(p in PIZZA)(how[p]=-v) >= buy[v]);
12  constraint forall(v in VOUCHER)(sum(p in PIZZA)(how[p]=-v) <= used[v]*buy[v]);
13  constraint forall(v in VOUCHER)(sum(p in PIZZA)(how[p]=v) <= used[v]*free[v]);
14  constraint forall(p1, p2 in PIZZA)((how[p1] < how[p2] /\ how[p1]= -how[p2])
15                                    -> price[p2] <= price[p1]);
16  int: total = sum(price);
17  var 0..total: objective = sum(p in PIZZA)((how[p] <= 0)*price[p]);
```

The first 5 lines introduce the parameters: lines 1 and 3 introduce n and m, respectively, line 2 introduces an array for the prices of the pizzas, while vouchers are introduced via two arrays in lines 4 and 5, where the ith voucher a/b is represented as `buy[i]/free[i]`. The next 3 lines define the variables: line 9 defines an array of vouchers, where `used[v]` is true iff voucher `v` was used. Line 8 defines an array of pizzas, where `how[p]` has value `v` if pizza `p` was free thanks to voucher `v`, has value `0` if `p` was paid for and not used in any voucher, and has value `-v` if `p` was paid for and used to get free pizzas with voucher `v`.

Constraints start in line 11, which states that if voucher `v` was used (`used[v]` holds), then the total number of pizzas bought and assigned to `v` must be greater than or equal to the number of pizzas required by it (`buy[v]`). The constraint in line 12 states similar information but in the opposite direction: the total number of pizzas bought

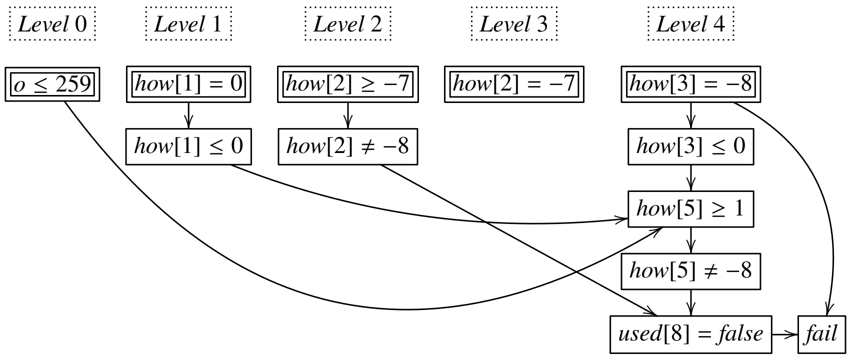

Fig. 2. Part of the implication graph for `freepizza`. Decision literals are double boxed.

and assigned to voucher `v` must be less than or equal to `used[v]*buy[v]`. Together they constrain the total number of pizzas bought for `v` to be equal to `buy[v]`, if used. The constraint in line 13 states that the total number of free pizzas obtained thanks to voucher `v` must be smaller than or equal to the number of free pizzas allowed by `v` if used (`used[v]*free[v]`). The last constraint is in line 14 and states that if there are two pizzas `p1` and `p2` assigned to the same voucher with `p2` being free and `p1` being paid for (given `how[p1] < how[p2]` and `how[p1] = -how[p2]`), then the price of `p2` must be lower than or equal to that of `p1`. Finally, the objective function is defined as the sum of the prices of the pizzas that are bought. Figure 1 shows a search tree for the execution of the model using Chuffed with the following input data:

```
n = 5;
price = [17, 98, 76, 36, 69];
m = 8;
buy = [4, 4, 1, 4, 2, 1, 1, 3];
free = [2, 4, 1, 1, 4, 2, 3, 3];
```

The third branch of the tree shows a restart after bound 259 has been established for the `objective`. Note that during the MiniZinc compilation process variable names are modified and, thus, variables `how_i` and `used_j` in the tree correspond to variables `how[i]` and `used[j]` in the model, respectively. Figure 2 shows part of the implication graph used to derive a nogood after the failure caused by search decision `how[3]=-8`. The failure set is {`how[3]=-8`, `used[8]=false`}. Since the nogood has two literals belonging to the current decision level (level 4), the last literal is reduced obtaining {`how[3]=-8`,`how[2]≠-8`,`how[5]≠-8`}. Since this set still has two literals belonging to the current decision level, literal `how[5]≠-8` is reduced to obtain {`how[3]=-8`,`how[2]≠-8`,`how[5]≥1`}. This reduction process continues until only one literal remains at the current decision level, yielding the 1UIP nogood {`objective≤259`, `how[1]≤0`,`how[2]≠-8`,`how[3]=-8`}. This nogood is negated and added to the search as clause {`objective>259`, `how[1]>0`, `how[2]=-8`, `how[3]≠-8`}, which is interpreted as the disjunction of its literals and prevents the same failure from recurring. After adding the clause, the search backjumps to level 2, as the nogood has no literal at level 3. This jump indicates that the decision at level 3 (`how[2]=-7`) is unrelated to the failure.

3 Exploring the most effective learnt clauses

The merging and replaying techniques mentioned above are useful when trying to understand the reasons for the success (or failure) of LCG solvers over traditional CP solvers. To remove the confounding effect of the different search orders, we can execute the LCG solver Chuffed first and replay its recorded decisions to execute the same model and search with Gecode [11], an efficient traditional CP solver. In general, for most constraints Gecode implements the same or stronger level of propagation as Chuffed. Therefore, the main differences when replaying the search in the form indicated above come from the clauses learnt by Chuffed. These clauses might help Chuffed (a) determine a failure earlier in the search, and/or (b) backjump further up than the parent node. In both cases, Gecode might perform extra search and, after merging the trees, those nodes will be displayed as pentagons by the tree visualisation.

We have modified Chuffed and the profiler to provide and visualise, respectively, extra information that is particularly useful when merging the replayed execution of an LCG solver by a CP solver. In particular, for each pentagon representing a failure node in the LCG execution, we can now compute and display the learnt clauses that helped to cause this failure. We refer to the number of pentagons to which a learnt clause contributed as *activity*, and measure its effectiveness in terms of search reduction, i.e., in terms of the number of nodes that Gecode explores and Chuffed does not thanks to the addition of the clause. Since several clauses can contribute to a failure, our *reduced search* measure divides the total number of nodes by the number of clauses involved.

This information is collated at the end of the execution and shown to the user in table form (see Table 1 for an example). We used this method to explore Chuffed's behaviour on three different problems. As illustrated below, the information shown by our tables can lead to insights that result in effective model transformations.

3.1 First case study: `freepizza.mzn`

The first problem we explored combines the `freepizza` model from the MiniZinc Challenge 2015 as introduced in Example 1, with the following input data:

```
n = 10;
price = [70, 10, 60, 65, 30, 100, 75, 40, 45, 20];
m = 4;
buy = [1, 2, 3, 3];
free = [1, 1, 2, 1];
```

We executed the problem using Chuffed, replayed its search using Gecode, merged their execution trees and explored the most effective learnt clauses in terms of reduced search and activity. Table 1 shows the top 10 clauses sorted by reduced search. We first concentrated on some of the shorter clauses, like `how[5]` $\neq$ `-3, how[3]` $\neq$ `3`, which is ranked number four and states that pizza 3 cannot be obtained for free with voucher 3 by buying pizza 5 and assigning it to this voucher. This is a direct consequence of the constraint in line 14 and the fact that pizza 3 (cost 60) is more expensive than pizza 5 (cost 30). This helped us understand the longer clauses and realise that some of them were weaker (and more complex) than they should. Consider, for example, the top clause, which captures information about the relationship between obtaining pizza 6 with vouchers 1 or 2 (as `how[6]` $\leq$ `0, how[6]` $\geq$ `3` indicates that `how[6]`

Table 1. Most effective learnt clauses in `freepizza`

Rank	Activity	Reduced Search	Clause
1	159	3425	how[1] = -1 how[2] = -1 how[3] = -1 how[4] = -1 how[5] = -1 how[1] = -2 how[2] = -2 how[3] = -2 how[4] = -2 how[5] = -2 how[6] ≤ 0 how[6] ≥ 3
2	176	2068	how[7] ≤ 2 how[7] ≥ 4 how[1] ≠ -3 how[1] ≥ -2
3	34	1712	how[4] ≠ 3 how[1] = -3 how[2] = -3 how[3] = -3 how[4] = -3
4	8	1636	how[5] ≠ -3 how[3] ≠ 3
5	8	1636	how[8] ≠ -3 how[3] ≠ 3
6	8	1636	how[9] ≠ -3 how[3] ≠ 3
7	8	1636	how[10] ≠ -3 how[3] ≠ 3
8	143	1489	how[6] ≤ 2 how[6] ≥ 4 how[1] ≠ -3 how[1] ≥ -2
9	25	1404	how[5] ≠ -3 how[4] ≠ 3 how[4] ≤ 2
10	24	1403	how[10] ≠ -3 how[4] ≠ 3

cannot be 1 or 2), and buying pizzas 1, 2, 3, 4, and 5, assigning them to these vouchers. It is clear by the input data that pizza 6 is more expensive than any other pizza and, thus, it cannot be obtained for free with any voucher (not just 1 and 2) and must be paid for. Therefore, the clause should be strengthened by expressing it as `how[6]`≤ `0`. It was surprising to realise that this simple fact (and its cousin: the cheapest pizza cannot be used to obtain any other pizza for free) was not being learnt by the solver. This interesting insight reinforced the usefulness of studying the learnt clauses to better understand the information learnt (or not learnt). While the learnings (`how[6]`≤ `0` and `how[2]`≥ `0`) were instance specific, the same ideas can be stated in a generic way and used as redundant constraints in the model:

```
% the most expensive pizza can never be bought with a voucher
constraint forall(p in PIZZA)
    (if forall(o in PIZZA where o != p)(price[p] > price[o])
       then how[p] <= 0 else true endif);
% the cheapest pizza can never be used with a voucher
constraint forall(p in PIZZA)
    (if forall(o in PIZZA where o != p) (price[p] < price[o])
       then how[p] >= 0 else true endif);
```

where != represents disequality. Of course, these redundant constraints will be vacuous if there is no single most expensive/cheapest pizza.

Another surprise was the fact that while many of the top clauses (4 to 7) were direct consequences of a single constraint (the one in line 14), learning them allowed Chuffed to avoid exploring significant amounts of nodes when compared to Gecode. We expected Gecode to also avoid exploring them by direct propagation. This indicated that the constraint was not propagating as strongly as expected. Upon inspection, it became clear that the `how[p1] < how[p2]` part of the constraint could be replaced by `how[p2] > 0`, indicating `p2` is free. This is clearly stronger information and connects with the way the objective function is expressed, thus allowing stronger propagation when the objective is bounded. The modified constraint is:

```
constraint forall(p1,p2 in PIZZA)((how[p2]>0/\how[p1]= -how[p2])
                                  -> price[p2] <= price[p1]);
```

Table 2. Aggregate Results for Free Pizza over a set of random instances (relative)

		Models Ratio	GeoMean(time)	Median(time)	GeoMean(fails)	Median(fails)
Fixed Search	Chuffed	redundant/original	0.4885	0.5497	0.5186	0.5737
		final/redundant	0.7746	0.7905	0.9159	0.9368
		final/original	0.3784	0.4152	0.9368	0.5199
	Gecode	redundant/original	0.0904	0.1250	0.0925	0.1234
		final/redundant	0.0569	0.0786	0.0435	0.0461
		final/original	0.0051	0.0056	0.0040	0.0042
Free Search	Chuffed	redundant/original	0.7039	0.7162	0.7625	0.7426
		final/redundant	0.8228	0.8070	0.8872	0.8944
		final/original	0.5791	0.5830	0.6765	0.6876
	Gecode	redundant/original	0.1526	0.1468	0.1493	0.1459
		final/redundant	0.7205	0.7330	0.7991	0.8104
		final/original	0.1100	0.1050	0.1193	0.1187

To assess the model changes we randomly generated 100 input data files and measured the solving time using Gecode and Chuffed with fixed search (as specified in the original model) and with free search. Since we aimed to solve all instances to completion within a reasonable amount of time (not too easy, not too difficult) for both solvers, we generated the input data with between 2 to 10 vouchers for Gecode and 6 to 10 for Chuffed, each voucher requiring 1 to 4 pizzas to be paid for and allowing customers to get 1 to 4 pizzas for free. We also used 9 to 10 pizzas for Gecode with fixed search, 12 to 13 for Gecode with free search, and 15 to 16 for Chuffed. We excluded from the final results any problem data file that, for a given solver and search, was solved in under one second for all models. Thus, we used 74 and 80 data files for Gecode with fixed and free search, respectively, and 98 and 95 data files for Chuffed.

Aggregated results for these data files are shown in Table 2, which compares the performance of the two solvers in terms of execution time and number of failures using three models: the original one, the one obtained by adding the two redundant constraints, and the final one obtained by also modifying the constraint in line 14 as indicated above. Clearly, our modifications improved the performance of both solvers (as all numbers are below 1), with the results being particularly significant for Gecode with fixed search, where the difference reaches two orders of magnitude. Detailed results are presented in Figure 3, where each dot shows the solving time for a given data file using the original and the final models in the horizontal and vertical axes, respectively. The scatter plots show that the vast majority of the instances lie below the identity line and, thus, that our final model consistently performs better than the original one.

3.2 Second case study: `radiation.mzn`

The second problem we explored is the **intensity-modulated radiation therapy (IMRT) problem** [2], where radiation is given through repeated exposures of a device that delivers a rectangular field of radiation of uniform intensity. This rectangular field is shaped

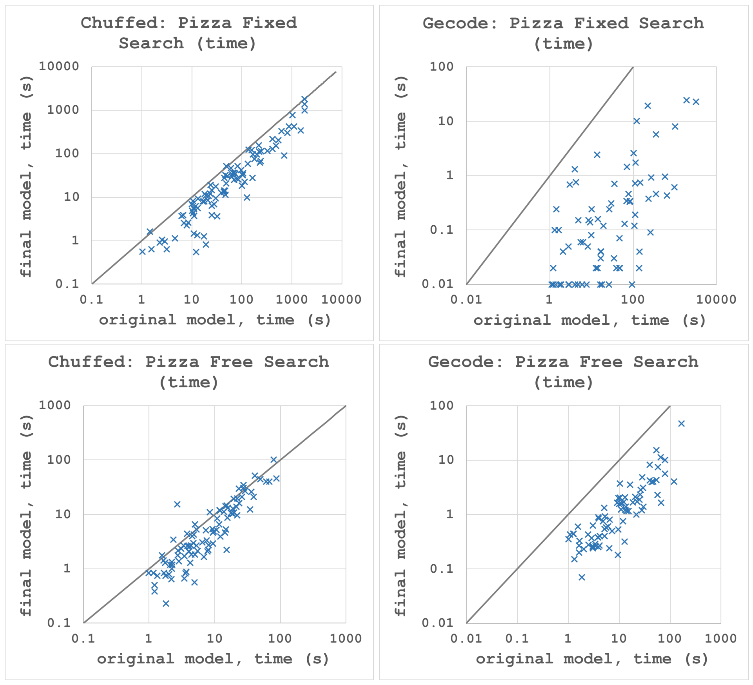

Fig. 3. Execution time of original and improved pizza models (logarithmic scale)

using horizontal lead rods positioned at the left and right of the rectangle, and which can slide laterally to block the radiation. In each exposure, the rods are moved into a given position, the radiation source switched on for a specified length of time and then switched off, to move to a new position. The model we studied is the one used in the MiniZinc Challenge 2015, where the input data is an $m \times n$ matrix `Intensity` of non-negative integers, where `Intensity[i,j]` represents the total amount of exposure that the cell in row i, column j should receive. The problem is to find a decomposition of the matrix into a positive linear combination of binary matrices, each with the consecutive-ones property (i.e., all 1s in any row are consecutive), where the 0s represent the part of the row occluded by the rods and the 1s the part that exposes radiation. The model is:

```
int: m;   % Rows
int: n;   % Columns
set of int: Rows    = 1..m;
set of int: Columns = 1..n;
array[Rows, Columns] of int: Intensity; % Intensity matrix
set of int: BTimes = 1..Bt_max;
int: Bt_max   = max(i in Rows, j in Columns) (Intensity[i,j]);
int: Ints_sum = sum(i in Rows, j in Columns) (Intensity[i,j]);

var 0..Ints_sum: Beamtime; % Total beam-on time
var 0..m*n: K; % Number of shape matrices
        % N[b] is the number of shape matrices with associated beam-on time b
array[BTimes] of var 0..m*n: N;
        % Q[i,j,b] is the number of shape matrices with associated beam-on time
        % b that expose cell (i,j)
array[Rows, Columns, BTimes] of var 0..m*n: Q;

```

```
18  constraint
19          Beamtime = sum(b in BTimes) (b * N[b])
20          /\
21          K = sum(b in BTimes) (N[b])
22          /\
23          forall(i in Rows, j in Columns)
24                  ( Intensity[i,j] = sum([b * Q[i,j,b] | b in BTimes]) )
25          /\
26          forall(i in Rows, b in BTimes)
27                  ( upper_bound_on_increments(N[b], [Q[i,j,b] | j in Columns]) );
28
29  predicate upper_bound_on_increments(var int: N_b, array[int] of var int: L) =
30          N_b >= L[1] + sum([ max(L[j] - L[j-1], 0) | j in 2..n ]);
31
32  int: obj_min = lb((m*n + 1) * Beamtime + K);
33  int: obj_max = ub((m*n + 1) * Beamtime + K);
34  var obj_min..obj_max: objective = (m*n + 1) * Beamtime + K;
```

The first 7 lines introduce the parameters of the problem: lines 1 and 2 introduce *m* and *n*, respectively, line 5 introduces the intensity matrix, line 7 computes in `Bt_max` the maximum intensity value in the matrix, and in `Ints_sum` the sum of all intensity values in the matrix, which is an upper bound to the total amount of time the radiation beam will have to be on. The next lines introduce the variables of the problem: line 10 defines the total beamtime `Bt_max` for the solution, line 11 defines the total number `K` of binary matrices in the solution (which has `m`$\times$ `n` as upper bound), line 13 defines vector `N`, where variable `N[b]` is the number of matrices with the same beamtime `b`, and line 16 defines array `Q`, where variable `Q[i,j,b]` is the number of binary matrices with beamtime `b` that expose cell `(i,j)` to radiation.

Constraints start on line 19, which states that the total beamtime is the result of adding the beamtime used for every binary matrix. Line 21 states that the total number of matrices is the result of adding those used for every beamtime. Line 23 states that the intensity required by each cell `(i,j)` in the intensity matrix must be achieved by the solution, that is, it must be equal to the sum of beamtimes for each of the binary matrices that expose that cell. Finally, line 26 states the consecutive-ones property of the binary matrices by ensuring that for every beamtime `b` and every row `i` of `Q[i,j,b]`, the number of times the intensity increases from a column `j-1` to the next `j`, is equal or less than the number of binary matrices with that beamtime `N[b]`.

Table 3 shows the 5 most effective learnt clauses obtained by executing the radiation model with the following input:

```
m = 9;  n = 9;  % rows and columns
Intensity = [| 4,  8, 11,  2,  5,  7,  1, 10,  4 |
              11,  4,  4,  5,  1,  8,  9,  3,  9 |
               2,  9,  6,  2,  4,  1,  5,  2,  6 |
              11,  9,  8,  9,  3,  2, 11,  6,  7 |
               2,  8, 11,  2, 10,  5,  5,  4,  5 |
               5,  9,  8,  1,  6,  3,  5, 11,  5 |
               ...
               7,  1,  6, 10,  0,  8,  1,  0,  0 |];
```

The top clause in Table 3 states that there should be a matrix that exposes cell [2,4] for a beamtime of 1, 3 or 5. This is because the input data requires the amount of radiation received by cell [2,4] to add up to exactly 5 units, which is an odd number. Thus, there needs to be at least one matrix with an odd beamtime. In particular, for 5 this requires a matrix with beamtime 1, 3, or 5, with anything longer than 5 resulting in the overexposure of the cell. This observation can be expressed in the model as follows:

Table 3. Most effective learnt clauses in `radiation`

Rank	Activity	Reduced Search	Clause
1	3	378	$Q_{2,4,1}\geq 1$ $Q_{2,4,5}\geq 1$ $Q_{2,4,3}\geq 1$
2	3	378	$Q_{2,6,1}\geq 2$ $Q_{2,6,1}\leq 0$ $Q_{2,6,7}\geq 1$ $Q_{2,6,8}\geq 1$ $Q_{2,6,5}\geq 1$ $Q_{2,6,2}\geq 4$ $Q_{2,6,4}\geq 2$ $Q_{2,6,3}\geq 1$
3	3	378	$Q_{2,1,1}\geq 1$ $Q_{2,1,5}\geq 1$ $Q_{2,1,3}\geq 1$ $Q_{2,1,9}\geq 1$ $Q_{2,1,10}\geq 1$ $Q_{2,1,11}\geq 1$ $Q_{2,1,8}\geq 1$ $Q_{2,1,7}\geq 1$ $Q_{2,1,3}\geq 2$
4	2	315	$Q_{2,9,2}\leq 0$ $Q_{2,9,3}\geq 1$ $Q_{2,9,4}\geq 2$ $Q_{2,9,5}\geq 1$ $Q_{2,9,6}\geq 1$ $Q_{2,9,7}\geq 1$ $Q_{2,9,8}\geq 1$ $Q_{2,9,9}\geq 1$ $(Q_{2,9,2} - Q_{2,8,2}) \geq 1$ $(Q_{2,9,1} - Q_{2,8,1}) \geq 2$
5	1	245	$Q_{2,9,4}\geq 1$ $Q_{2,9,3}\geq 2$ $Q_{2,9,1}\geq 5$ $Q_{2,9,7}\geq 1$ $Q_{2,9,8}\geq 1$ $Q_{2,9,9}\geq 1$ $Q_{2,9,6}\geq 1$ $Q_{2,9,5}\geq 1$ $(Q_{2,9,1} - Q_{2,8,1})\geq 2$ $(Q_{2,9,2} - Q_{2,8,2})\geq 3$ $(Q_{2,9,3} - Q_{2,8,3})\geq 1$

```
constraint
  forall(b in BTimes where b mod 2 = 1)
    (forall(i in Rows, j in Columns where Intensity[i, j] = b)
      (sum([Q[i,j,k] | k in 1..b where k mod 2 = 1]) > 0));
```

While adding this constraint reduces the amount of search space explored, the reduction is small (3.0% measured as median over 20 random instances), and is outweighed by the cost of propagating the extra constraints resulting in a 4.6% longer execution. This suggests that Chuffed's good performance on this problem is not due to the learnt clauses, but to its conflict analysis (as confirmed in Section 4).

3.3 Third case study: Golomb ruler

A Golomb ruler of size n is a set of n integer marks on an imaginary ruler, such that no two pairs of marks are the same distance apart. An optimal ruler is one with minimum length; i.e. the largest mark is to be minimised. The MiniZinc benchmarks set contains a model for finding such rulers[3], with two arrays of variables, one holding n integer variables (the `marks`) with domain $0..n^2$, and the other holding $\frac{n(n-1)}{2}$ integer variables (the `differences`) with domain $1..n^2$. This model is known to be difficult for learning solvers. Indeed, we find that Gecode is consistently faster than Chuffed on this model (see "Original Model" in Table 5). Nonetheless, Chuffed requires fewer failures to solve the problem and, thus, we decided to examine Chuffed's behaviour to see if we could learn something to help improve the model.

The top of Table 4 shows the 5 most effective clauses learnt by Chuffed while searching for $n = 10$. All these clauses are of the form `mark[i] >= n & mark[i+1] >= n+1 -> mark[i+2] >= n+3`, for some `i` and `n`. (Note that a clause of the form $\{A, B, C\}$ can be interpreted as $\neg B \wedge \neg C \rightarrow A$.) This indicates a missing constraint which the solver is effectively rediscovering. Looking again at the problem, we confirmed it was correct to add the following redundant constraint: `mark[i] + 3 <= mark[i+2]`, for all `i`. Clearly `mark[i+2]` is at least one more than `mark[i+1]`,

[3] https://github.com/MiniZinc/minizinc-benchmarks

Table 4. Most effective learnt clauses for Golomb Ruler (before and after the first modification)

	Rank	Activity	Reduced Search	Clause
Original	1	49	193	mark[6] ≥ 38, mark[5] ≤ 35, mark[4] ≤ 34
	2	6	170	mark[5] ≥ 18, mark[4] ≤ 15, mark[3] ≤ 14
	3	6	170	mark[5] ≥ 15, mark[4] ≤ 12, mark[3] ≤ 11
	4	5	168	mark[5] ≥ 22, mark[4] ≤ 19, mark[3] ≤ 18
	5	50	163	mark[6] ≥ 36, mark[5] ≤ 33, mark[4] ≤ 32
Modified	1	2	55	mark[6] ≥ 19, mark[4] ≤ 14, mark[3] ≤ 13
	2	3	55	mark[6] ≥ 18, mark[4] ≤ 13, mark[3] ≤ 12
	3	3	41	(mark[8] - mark[6] ≤ 5), (mark[8] - mark[6] ≥ 7), (mark[10] ≥ 55), (mark[6] ≤ 41), (mark[10] - mark[8] ≤ 5)

which is at least one more than `mark[i]`. Thus, `mark[i+2]` and `mark[i]` are at least two apart and, if so, both intermediate differences must be one, which is forbidden.

We added this redundant constraint to the model, re-executed the modified model and examined the 5 most effective learnt clauses (see bottom of Table 4). The first two follow the pattern `mark[i] >= n & mark[i+1] >= n+1 -> mark[i+3] >= n+5`. The third illustrates a property of connected differences: if the difference between `mark[i]` and `mark[j]` is e.g. 6, and the difference between `mark[j]` and `mark[k]` is at least 6, then the difference between `mark[i]` and `mark[k]` is at least 6+6+1=13. (The extra one appears for the same reason as above.) All these clauses refer to the idea that the distance between two marks that are m positions apart is equal to the sum of the inner distances, which are all different. As a result, this distance is at least as large as the sum of the arithmetic sequence of natural numbers, i.e., $\frac{m(m+1)}{2}$. In fact, by following this methodology we rediscovered a redundant constraint for this problem that was earlier discussed in [3].

The inclusion of this redundant constraint reduces the search effort required to solve the problem, both in number of nodes and in time (see "Improved Model" in Table 5). Interestingly, even the non-learning solver Gecode benefited from the constraint. This demonstrates how, even when learning solvers are not the strongest for a given problem, we can gain useful insights from examining their behaviour.

Table 5. Golomb Ruler Results

		Original Model		Improved Model	
	Size n	Time (s)	Number of Failures	Time (s)	Number of Failures
Chuffed	n = 10	1.99	20,912	1.49	19,343
	n = 11	72.36	307,957	54.25	288,071
	n = 12	616.2	2,329,959	512.63	2,254,206
Gecode	n = 10	0.74	23,463	0.57	19,928
	n = 11	15.81	374,886	12.09	321,419
	n = 12	147.00	3,002,474	117.83	2,656,663

4 Profiling statistics

In addition to computing and showing the most effective clauses in table form, we have modified the profiler to compute statistical data based on the information provided by the solvers. In particular, for a learnt clause $L \equiv \{l_1, \ldots, l_m\}$ the profiler can now display the following information:

– **Length**: the number m of literals in L.
– **Decision level**: the decision level at which the failure occurred.
– **Total number of variables**: the cardinality of the set $vars(L)$. Note that this is always less than or equal to the length. The number of variables can be much smaller than the length if variables appear in many literals.
– **Literal Block Distance**: number of decision levels where literals in L were inferred. Note that this can never be larger than the decision level for L. This measure has been shown to be useful in SAT problems [1].
– **Backjump destination**: the decision level of the node to which the search backjumped after learning L.
– **Backjump distance**: the distance between the decision level of the node in which L was learnt and that of the backjump destination. Note that if it is 1, the behaviour is similar to traditional CP backtracking.
– **Activity**: number of times L is involved in inferring failures later in the search.
– **Size reduction**: number of nodes avoided thanks to L being learnt (both in terms of the tree that would have been explored by a CP solver and in terms of backjumped ones). This measure requires a comparison with Gecode's execution.
– **Generation time**: point in time during the search at which L was learnt.

Example 2. The length of the clause {`objective>259`, `how[1]>0`, `how[2]=-8`, `how[3]≠-8`} found in Example 1 is 4, which is equal to its number of variables. Its decision level is 4, its literal block distance is also 4, its backjump destination is level 2, and its backjump distance is also 2.

As mentioned before, our aim is to determine whether any of this information could be used to explain the reasons behind good or bad performance and, thus, should be highlighted to users as possible markers for such behaviour. Figure 4 provides an example of the plots that display some (due to space limitations) of this statistical information. Each dot in each square represents a single clause. There is a scatter plot for each pair of attributes, arranged in a triangular matrix. The attributes shown are (in order from left to right and top to bottom) time at which the associated clause was learnt (in microseconds), decision level, backjump distance, literal block distance, backjump destination and raw activity. The plots along the diagonal show kernel density estimates of each attribute. For example, the plot shown in the fifth row and second column (denoted (5,2)) shows the backjump destination against the decision level, while plot (3,3) shows the overall distribution of backjump distances.

Let us consider Figure 4, which shows the statistics obtained for an instance of the `radiation` problem model with fixed search (left) and free search (right), with Chuffed performing much better for the fixed search. Let us compare this with Figure 5, which shows the statistics for two instances of the MiniZinc Challenge problems where Chuffed does not performs well: `cvrp` (left) and `opd` (right). There are a few interesting things to note. First, comparing the (6,1) plots, there is a relatively high level of activity

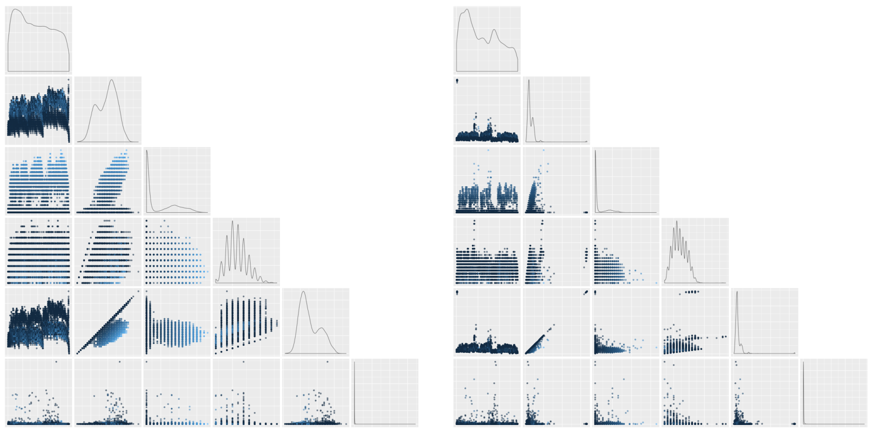

Fig. 4. Profiling plots for the `radiation` problem with Chuffed using fixed and free search.

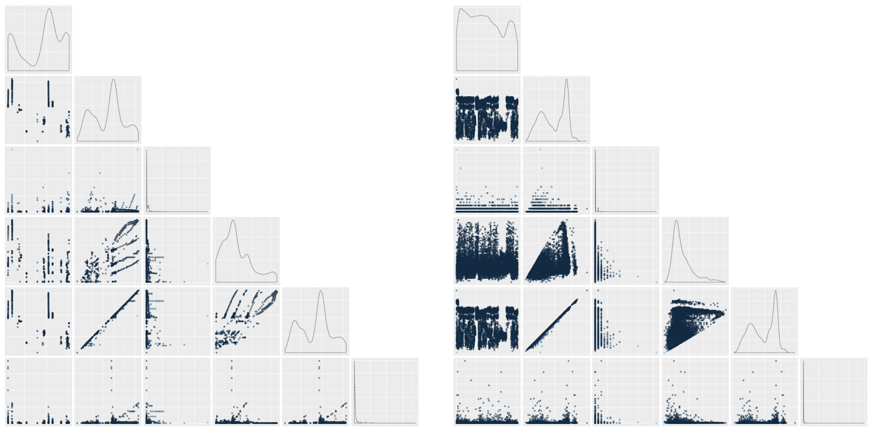

Fig. 5. Profiling plots for `cvrp` (left) and `opd` (right).

throughout the entire execution of `radiation`, while the level of activity in the other two problems is smaller and, in `cvrp`, only appears at particular points in time.

From the plots we can also observe the high-level behaviour of the search. For `cvrp` and `opd` the (5,2) plots shows quite a compact diagonal, indicating that the backjump destination level of most clauses is relatively similar to their decision level – in other words, the search does not backjump significantly. This is not the case for `radiation`, where there is a significant "bulge" below the diagonal.

The `radiation` problem on the left exhibits a related phenomenon, visible in plots (2,2) and (5,5), which show the distribution of decision level and backjump destination. In the fixed search, the peaks of these plots are the reverse of one another, a phenomenon unique among all the instances shown. This indicates that the search backjumps from a deeper level (the peak on the right) to a shallower level (the peak on the left). Indeed, the search is designed to encourage this behaviour; it fixes the variables in such a way that after the early "master" variables have been selected, the problem consists of wholly independent subproblems. Whenever one such subproblem fails, *all* subproblems can be discarded and the search returns to the master variables. This backjumping caused by conflict analysis is the reason that Chuffed performs well on this problem, as suggested in Section 3.2. The free search performs worse because this behaviour occurs less frequently. The plots of these statistics confirm that the specified fixed search is performing as expected.

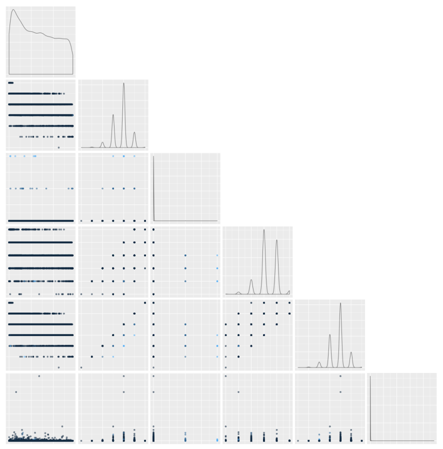
Fig. 6. Golomb ruler original model, $n = 11$

Finally, Figure 6 shows the profiling plot for the Golomb ruler problem examined in Section 3. We observe in the (3,3) plot that there is no significant backjumping during the search. This partially explains why Chuffed is slower than Gecode, and confirms that any benefit for the learning solver comes from the learnt clauses and not from any improvement to the search behaviour.

From these and other examples we have already identified some statistical markers of learning behaviour, including:

- Literal block distance being always close to the decision level: this indicates that failure explanations are poor and backjumping is likely to be minimal;
- Failure decision level being confined to a narrow range, especially deep in the tree: this is a clear indication of search making no progress;
- Deep decision levels coupled with low backjump distance and low learnt clause activity: this strongly suggests poor performance.

5 Conclusions

Learning solvers dramatically outperform traditional CP solvers on many problems, but their behaviour in practice is opaque and hard to understand. We have instrumented the learning solver Chuffed to give detailed information about its behaviour, so that it can be better understood. In particular, we have considered several case studies and shown how profiling leads to a better understanding of the solver's behaviour on each problem, and how the profiling information can lead directly to improvements of the model by either modifying its constraints or adding new redundant ones. One may argue that redundant constraints such as those derived in Section 3 could just as well be found without any profiling data. However, the method we show here allows the solver to tell us precisely the constraints it requires to reach its conclusion, avoiding the "guess and test" approach to model improvement.

This work prompts further analysis of learning behaviour. In particular, there is the possibility to include other features of learnt clauses, and to determine via machine learning techniques specific markers for good or bad performance. As well as providing feedback to the user, such indicators could be used to guide heuristics for solvers when performing autonomous search. It would also be interesting to apply the presented techniques to SMT solvers [9]. Further, while here we have focused on the statistical analysis of learned clauses, studying the clause graph structure, as for example in [8], can be insightful as well. The profiler and the modified versions of Chuffed and Gecode used in this work are available at `http://www.minizinc.org`.

Acknowledgements. We thank the anonymous reviewers who pointed to overlooked related work and provided useful comments. This research was partly sponsored by the Australian Research Council grant DP140100058.

References

1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st International Joint Conference on Artifical Intelligence. pp. 399–404. IJCAI'09 (2009)
2. Baatar, D., Boland, N., Brand, S., Stuckey, P.J.: CP and IP approaches to cancer radiotherapy delivery optimization. Constraints 16(2), 173–194 (2011)
3. Barták, R.: Effective modeling with constraints. In: Applications of Declarative Programming and Knowledge Management, pp. 149–165. Springer (2005)
4. Chu, G.G.: Improving combinatorial optimization. Ph.D. thesis, The University of Melbourne (2011)
5. Feydy, T., Stuckey, P.J.: Lazy Clause Generation Reengineered. In: Gent, I.P. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 5732, pp. 352–366. Springer (2009)
6. Marques Silva, J.P., Sakallah, K.A.: GRASP – a new search algorithm for satisfiability. In: Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design. pp. 220–227. ICCAD '96, IEEE Computer Society, Washington, DC, USA (1996), `http://dl.acm.org/citation.cfm?id=244522.244560`
7. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference. pp. 530–535. ACM (2001)
8. Newsham, Z., Lindsay, W., Liang, J.H., Czarnecki, K., Fischmeister, S., Ganesh, V.: SATGraf: Visualizing community structure in boolean SAT instances. `https://ece.uwaterloo.ca/~vganesh/EvoGraph/Home.html` (2014)
9. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Abstract DPLL and abstract DPLL modulo theories. In: Baader, F., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3452, pp. 36–50. Springer (2004)
10. Ohrimenko, O., Stuckey, P.J., Codish, M.: Propagation = Lazy Clause Generation. In: Bessiere, C. (ed.) Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 4741, pp. 544–558. Springer (2007)
11. Schulte, C., Tack, G., Lagerkvist, M.Z.: Modeling and programming with Gecode (2016), `http://www.gecode.org`
12. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.: Why Cumulative Decomposition Is Not as Bad as It Sounds. In: Gent, I.P. (ed.) Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 5732, pp. 746–761. Springer (2009)
13. Schutt, A., Stuckey, P.J., Verden, A.R.: Optimal carpet cutting. In: CP. pp. 69–84 (2011)
14. Shishmarev, M., Mears, C., Tack, G., de la Banda, M.G.: Visual search tree profiling. Constraints 21(1), 77–94 (2016)
15. Stuckey, P.J., Feydy, T., Schutt, A., Tack, G., Fischer, J.: The MiniZinc challenge 2008-2013. AI Magazine 35(2), 55–60 (2014)