Automatically Improving Constraint Models in Savile Row: Supplementary Material

Peter Nightingale[a], Özgür Akgün[a], Ian P. Gent[a], Christopher Jefferson[a], Ian Miguel[a], Patrick Spracklen[a]

[a]*School of Computer Science, University of St Andrews, St Andrews, Fife KY16 9SX, UK.*

Abstract

This technical report contains supplementary material for the journal paper *Automatically Improving Constraint Models in Savile Row* [8].

1. Encoding to SAT

Our goal is to investigate whether reformulations performed on a constraint problem instance are beneficial when the problem instance is solved by encoding to SAT and using a state-of-the-art SAT solver. To achieve this we need to ensure that the baseline encoding to SAT is sensible. Therefore we have used standard encodings from the literature such as the order encoding for sums [9] and support encoding [5] for binary constraints. Also we do not attempt to encode all constraints in the language: several constraint types are decomposed before encoding to SAT.

Encoding of CSP variables

The encoding of a CSP variable provides SAT literals for facts about the variable: $[x = a]$, $[x \neq a]$, $[x \leq a]$ and $[x > a]$ for a CSP variable x and value a. CSP variables are encoded in one of three ways. If the variable has only two values, it is represented with a single SAT variable. All the above facts (for both values) map to the SAT variable, its negation, *true* or *false*. If the CSP variable is contained in only sums, then only the order literals $[x \leq a]$ and $[x > a]$ are required. Using the language of the *ladder* encoding of Gent and Nightingale [6], we have only the ladder variables and the clauses in Gent and Nightingale formula (2). Otherwise we use the full ladder encoding with the clauses in formulas (1), (2) and (3) of Gent and Nightingale. Also, for the maximum value $\max(D(x))$ the facts $[x \neq \max(D(x))]$ and $[x < \max(D(x))]$ are equivalent so one SAT variable is saved. Finally, a variable may have gaps in its domain. Suppose variable x has domain $D(x) = \{1 \ldots 3, 8 \ldots 10\}$. SAT variables are created only for the 6 values in the domain. Facts containing values $\{4 \ldots 7\}$ are

Email addresses: pwn1@st-andrews.ac.uk (Peter Nightingale), ozgur.akgun@st-andrews.ac.uk (Özgür Akgün), ian.gent@st-andrews.ac.uk (Ian P. Gent), caj21@st-andrews.ac.uk (Christopher Jefferson), ijm@st-andrews.ac.uk (Ian Miguel), jlps@st-andrews.ac.uk (Patrick Spracklen)

mapped appropriately (for example $[x \leq 5]$ is mapped to $[x \leq 3]$). The encoding has $2|D(x)| - 1$ SAT variables.

Decomposition

The first step is decomposition of the constraints AllDifferent, GCC, Atmost and Atleast. All are decomposed into sums of equalities and we have one sum for each relevant domain value. For example to decompose AllDifferent($[x, y, z]$), for each domain value a we have $(x = a) + (y = a) + (z = a) \leq 1$. These decompositions are done after AC-CSE if AC-CSE is enabled (because the large number of sums generated hinders the AC-CSE algorithm), and before Identical and Active CSE [8, Sections 5.7 and 5.8].

The second step is decomposition of lexicographic ordering constraints. We use the decomposition of Frisch et al [3, Section 4] with implication rewritten as disjunction, thus the conjunctions of equalities in Frisch et al become disjunctions of disequalities. This decomposition is also done after AC-CSE and before Identical and Active CSE. However, if AC-CSE is switched on, we (independently) apply AC-CSE to the decomposition, thus extracting common sets of disequalities from the disjunctions.

The third step occurs after general flattening is completed. The constraints min, max and element are decomposed. For $\min(V) = z$ we have $\exists i : V[i] = z$ and $\forall i : z \leq V[i]$. Max is similar to min with $\leq$ replaced by $\geq$. The constraint $\mathrm{element}(V, x) = z$ becomes $\forall i : (x \neq i \vee V[i] = z)$.

Encoding of Constraints

Some simple expressions such as $x = a$, $x \leq a$ and $\neg x$ (for CSP variable x and value a) may be represented with a single SAT literal. Savile Row has an expression type named SATLiteral. Each expression that can be represented as a single literal is replaced with a SATLiteral in a final rewriting pass before encoding constraints. SATLiterals behave like boolean variables hence they are transparently included in any constraint expression that takes a boolean subexpression.

For sums we use the order encoding [9] and to improve scalability sums are broken down into pieces with at most three variables. Sum-equal constraints are split into sum-$\leq$ and sum-$\geq$ before encoding. For other constraints we used the standard support encoding wherever possible [5]. Binary constraints such as $|x| = y$ use the support encoding, and ternary functional constraints $x \diamond y = z$ (e.g. $x \times y = z$) use the support encoding when z is a constant. Otherwise, $x \diamond y = z$ are encoded as a set of ternary SAT clauses: $\forall i \in D(x), \forall j \in D(y) : (x \neq i \vee y \neq j \vee z = i \diamond j)$. When $i \diamond j$ is not in the domain of z, the literal $z = i \diamond j$ will be false. Logical connectives such as $\wedge, \vee, \leftrightarrow$ have custom encodings and table constraints use Bacchus' encoding [2] (Sec.2.1).

2. Additional Experiments

In this section we present additional experimental results not given in the main paper [8].

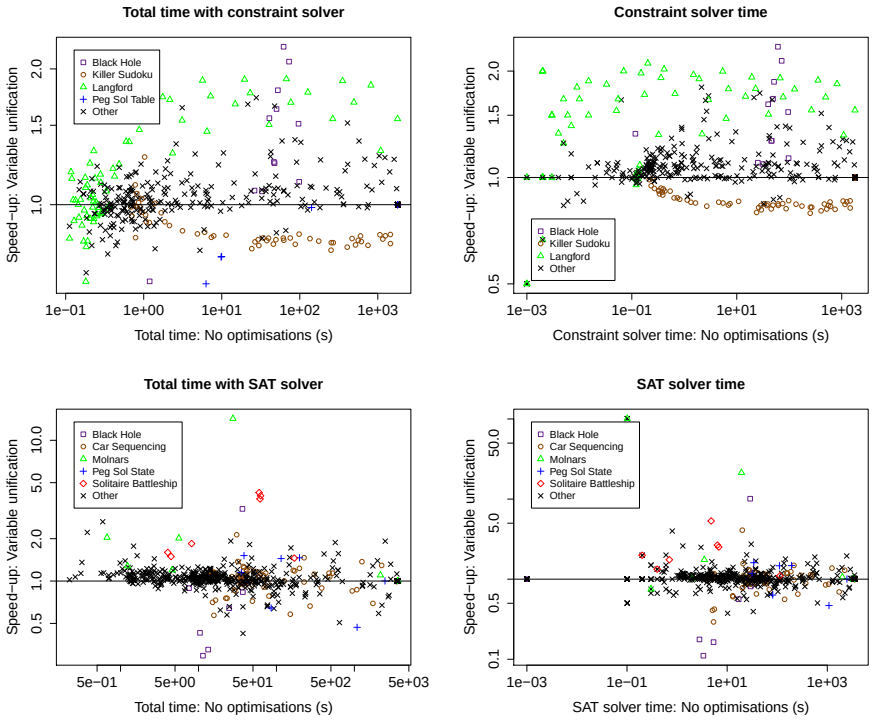

Figure 1: VarUnif vs Simp. In the upper plots the solver is Minion, in the lower plots it is SAT (Lingeling). Plots on the left show total time, plots on the right show only solver time. In each case, the y axis is the speed up quotient caused by the optimisation, so points above 1 on the y axis exhibit a speed improvement. The x axes are time (in seconds) for the variant without the optimisation. Both axes are logarithmic.

2.1. Experiment A: Variable unification

We compare variable unification to the baseline configuration of Savile Row where only simplifiers are switched on (VarUnif vs Simp). Figure 1 plots our findings. We plot all problem classes together and pick out a few problem classes that are particularly interesting using a distinct colour and plot symbol.

Considering only constraint solver time, variable unification exhibits some benefit for the constraint solver, particularly on Langfords Problem and Black Hole Solitaire. Given that the constraint solver has a static variable and value ordering (and variable unification does not change the variable ordering), variable unification is unlikely to reduce the size of the search. Its main potential benefit is to speed up search by reducing overhead.

The Killer Sudoku problem is an exception. This problem class reveals an unusual feature of Minion: using a single variable type in a constraint can speed up propagation because the propagator is specialised for the variable type using a C++ template, allowing function calls to be inlined. Variable unification replaces some decision variables

with constants in allDiff constraints, and constants are a distinct variable type. This results in a slowdown of (at most) 21% on Killer Sudoku 16x16.

Considering total time with the constraint solver (the upper left plot of Figure 1), variable unification can be beneficial and also it can add a small overhead to Savile Row. For example, the Peg Solitaire Table class is slowed down: the smallest speed-up quotient is 0.67. Peg Solitaire Table has a large set of table constraints that are simplified (by selection and projection of the tables) following variable unification. Running the table simplifier is a potential cause of the slow-down.

For the SAT solver, variable unification changes the CNF formulation of the problem and can therefore entirely change the search order. This can cause large effects in both directions, even within a problem class. For example, three instances of Black Hole solitaire exhibit a substantial slow-down of between 5 and 10 times, and one instance shows a speed-up of just over 10 times (SAT solver time only).

The geometric mean speed up quotient (of total time) when using the CP solver is 1.046 with confidence interval [1.027, 1.066], and for the SAT solver it is 1.061 with confidence interval [1.038, 1.086]. In summary, variable unification is slightly beneficial when producing output for either solver and these results are statistically significant. Variable unification also enables other reformulations to trigger (in particular, AC-CSE) and it will be switched on for all the following experiments.

2.2. Experiment B: Domain filtering

The main paper describes standard and extended domain filtering [8, Section 5.4], and evaluates standard domain filtering together with variable unification. Here we compare standard domain filtering combined with variable unification to variable unification alone (DomFilt vs VarUnif). Figure 2 plots the results.

Considering constraint solver time only, it is clear that some problem classes benefit from domain filtering (Black Hole, Killer Sudoku, and Solitaire Battleship). The high outlier is an instance of Magic Sequence that was entirely solved (i.e., all variables assigned) by singleton bound consistency. In this case the work involved in solving the instance has been moved from the back-end solver to the domain filtering step, and no time is saved overall.

When including Savile Row time, the picture is different. Killer Sudoku, BIBD and BIBD Implied are only improved for the harder instances. The geometric mean speed-up (total time) is 0.920 with confidence interval [0.903, 0.937]. However, there does appear to be an upward trend in Figure 2 (upper left) where harder instances are more likely to benefit from domain filtering.

For the SAT solver (taking solver time only) we find that some difficult problem instances become substantially easier when domain filtering is applied. Magic Sequence and Send More Money are completely solved by domain filtering. Most Solitaire Battleship instances become much smaller and faster to solve. Also there are several Killer Sudoku instances that become trivial for the SAT solver.

When Savile Row time is included, there are more modest speed gains for some instances of Killer Sudoku, both instances of Magic Sequence, and Send More Money. Solitaire Battleship instances are slightly improved. The geometric mean speed-up (total time) is 1.250 with confidence interval [1.190, 1.316].

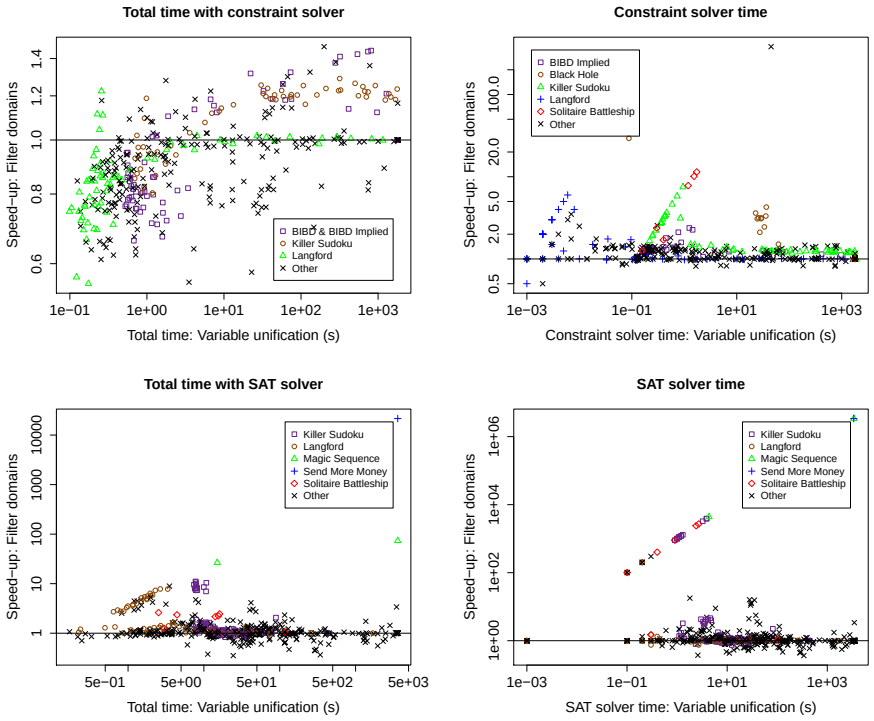

Figure 2: DomFilt vs VarUnif. In the upper plots the solver is Minion, in the lower plots it is SAT (Lingeling). Plots on the left show total time, plots on the right show only solver time. Axes are the same as Figure 1.

Although domain filtering does not provide a clear benefit for both solvers, it enables AC-CSE to substantially improve the formulation of some problem classes (as shown in the main paper [8]). Hence domain filtering will be switched on for all subsequent experiments.

2.3. Experiment C: Extended domain filtering

In addition to standard domain filtering (of user-defined variables), *extended* domain filtering transfers filtered domains of auxiliary variables that may have different names in the first and second tailoring process. To achieve this we link each auxiliary variable to an expression *e* that it represents, using the following two data structures (implemented with hash tables):

AuxToE A function from auxiliary variable names to expressions they represent in the current tailoring process.

EToDomain A function from expressions to filtered domains.

When an auxiliary variable a is created representing an expression e in the first tailoring process, $a \mapsto e$ is added to **AuxToE**. After calling the solver and reading in a filtered domain d for a, the expression e is retrieved from **AuxToE** then $e \mapsto d$ is stored in **EToDomain**. In the second tailoring process, **AuxToE** is cleared and re-populated to ensure it always contains the set of auxiliary variables for the current tailoring process.

Expressions stored or used as keys in **AuxToE** and **EToDomain** must not contain auxiliary variables because the names are not stable from one tailoring process to the next. Whenever a new expression e is stored or used as a key in either hash table, **AuxToE** is used to replace all auxiliary variables in e with expressions containing only the `find` decision variables.

In the second tailoring process when an auxiliary variable a is required for e, the filtered domain d_1 is retrieved from **EToDomain**. Bounds are obtained for e (as described in the main paper [8, Section 3.2]) and the final domain for the auxiliary variable a is $d_1 \cap \{\lfloor e \rfloor \dots \lceil e \rceil\}$.

Expressions evolve during a tailoring process by simplification, variable unification and normalisation [8, Sections 3.3, 5.4, 5.6]. Whenever these transformations are applied to the model they are also applied to expressions in **EToDomain** to maintain syntactic equality between expressions in the model and those in the hash table.

In this method each auxiliary variable is associated with exactly one expression. In fact an auxiliary variable may represent a set of expressions S_1 (e.g. when using Active CSE). In this case it is stored once in **AuxToE** with an arbitrarily chosen expression from S_1. In the second tailoring process, when an auxiliary variable a is required for some set of expressions S_2, each element of S_2 is looked up in **EToDomain**.

Leo and Tack proposed *variable paths* as a unique identifier of auxiliary variables that is stable across both tailoring processes in MiniZinc [7]. A variable path is a string containing all the relevant state of the MiniZinc interpreter when the auxiliary variable was created. A variable path includes the names and locations of functions that have been called and the names and values of loop (comprehension) variables. Using this identifier they store filtered domains of auxiliary variables created in the

first tailoring process and retrieve them during the second tailoring process. Both the approach in Leo and Tack and ours essentially use canonical names to link variables in the first and second tailoring processes. The main difference between the two is the strength of filtering: Leo and Tack apply the standard propagation level of the Gecode solver [4], whereas we apply SACBounds which will be more powerful (ignoring minor differences between Gecode and Minion) and time-intensive.

We compare extended domain filtering to standard domain filtering. When either type of domain filtering is switched on, the time reported for Savile Row includes the call to Minion to perform SACBounds. Minion is called identically for both types.

As noted in the main paper [8, Section 5.2] Minion (when used as a backend solver) always applies SACBounds directly before search. Therefore extended domain filtering has two potential benefits: to trigger other reformulations within Savile Row, or to improve the efficiency of Minion without changing the search tree (for example, by reducing the time taken for the initial SACBounds pass).

Our results are plotted in Figure 3. Looking at constraint solver time only, we can see that extended domain filtering can be beneficial, notably for Golomb Ruler and some instances of BIBD and BIBD Implied. When Savile Row time is included, only Golomb Ruler shows a clear benefit.

The model of Golomb Ruler contains the constraints `ruler[j] - ruler[i] != ruler[l] - ruler[k]` for all i, j, k, l where $i < j$, $k < l$ and $[i, j] <_{lex} [k, l]$. The ruler variables are ordered such that `ruler[j]>ruler[i]` and `ruler[l]>ruler[k]`. Savile Row introduces auxiliary variables for the subexpressions of the form `ruler[j]-ruler[i]`. When all forms of CSE are switched off (as they are in this experiment) multiple auxiliary variables are created for each expression of the form `ruler[j]-ruler[i]` where $i < j$. For the instance of length 9, there are 980 of these auxiliary variables (representing only 36 distinct expressions) with domain size of about 130, and with roughly half the values being negative. Extended domain filtering removes (at least) the values below 1. It does not trigger any other reformulation in Savile Row, and does not reduce search. The benefit comes entirely from reducing overhead in the backend solver.

When aggregation is switched on, the not-equal constraints of Golomb Ruler are collected into an allDifferent constraint that contains each difference once, solving the problem caused by the large number of auxiliary variables and eliminating the improvement made by extended domain filtering.

With the constraint solver, the geometric mean speed-up quotient (of total time) is 0.995 with confidence interval [0.987, 1.004], so the difference is not statistically significant.

When encoding to SAT, extended domain filtering reduces the numbers of both SAT variables and clauses without fundamentally changing the formulation of the problem, so one might expect the performance of the SAT solver never to be degraded. Figure 3 (lower) plots the results. In fact on some instances Lingeling's performance is degraded. For example the BIBD class is 0.33 to 1.41 times faster, and BIBD Implied ranges from 0.42 to 2.30 times faster (total time). Overall it is not clear whether extended domain filtering improves performance of the SAT solver. The geometric mean speed-up quotient (of total time) is 0.981 with confidence interval [0.972, 0.989].

In summary extended domain filtering provides no clear benefit and it will not be

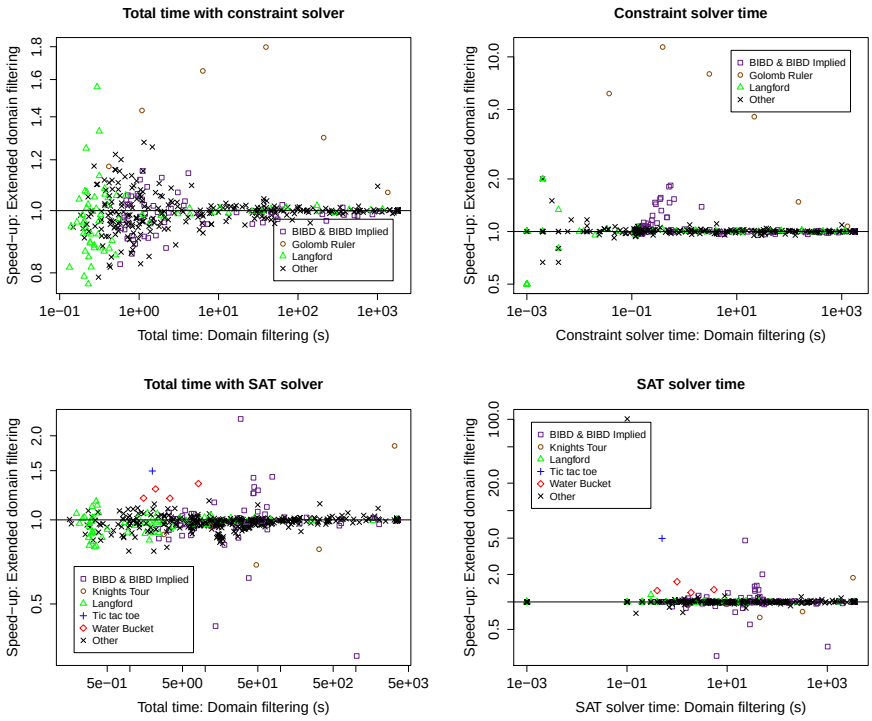

Figure 3: Extended domain filtering vs domain filtering (EDomFilt vs DomFilt). In the upper plots the solver is Minion, in the lower plots it is SAT (Lingeling). Plots on the left show total time, plots on the right show only solver time. Axes are the same as Figure 1.

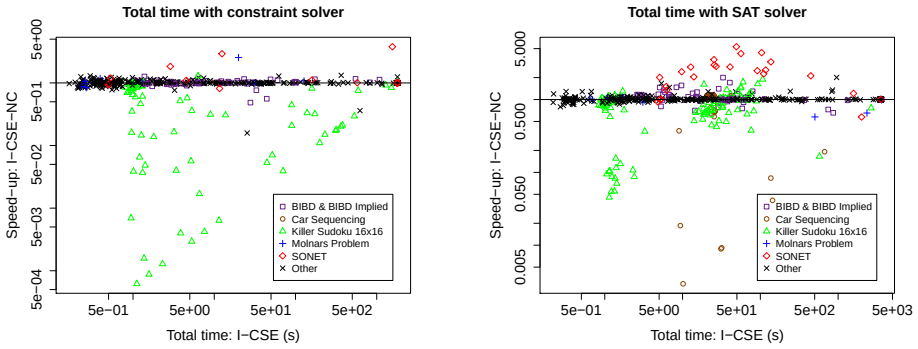

Figure 4: Comparing I-CSE-NC to I-CSE (total time) with the CP solver (left) and the SAT solver (right).

enabled in subsequent experiments.

2.4. Experiment D: I-CSE, I-CSE-NC and I-CSE-Iter

In this section we will compare I-CSE (as proposed by Araya et al [1]) with variants I-CSE-NC and I-CSE-Iter. I-CSE-NC is a version of I-CSE that does not copy expressions, so it cannot exploit both of a conflicting pair of AC-CSs and in some cases it generates a smaller set of constraints than I-CSE. Figure 4 compares total time of I-CSE-NC to I-CSE. With the CP solver I-CSE-NC never reduces search. I-CSE performs less search than I-CSE-NC on 100 instances (of 8 problem classes), and this translates into faster solving for Killer Sudoku (with both solvers) and Car Sequencing (with the SAT solver). The more compact formulation of I-CSE-NC can be better, notably on the SONET problem for both solvers. The geometric mean speed-up of total time with the CP solver is 0.845 (with confidence interval [0.785, 0.904]) and with the SAT solver is 0.952 (with confidence interval [0.909, 0.995]). Overall it seems that I-CSE-NC is inferior to I-CSE.

We proposed I-CSE-Iter to address one of the shortcomings of I-CSE: that some AC-CSs are not exploited by I-CSE (for example, an intersection of three sums that is not the intersection of any pair of sums). I-CSE-Iter first calls I-CSE on the original constraints, then repeatedly calls I-CSE on the AC-CSs extracted by the previous call until no more AC-CSs are found. Figure 5 compares I-CSE-Iter to I-CSE. For both solvers I-CSE-Iter improves Killer Sudoku 16×16 (where a clue, a row or column, and a subsquare frequently intersect). With the CP solver X-CSE solves 76 and I-CSE-Iter solves 78 instances, and the geometric mean speed-up of X-CSE (total time) is 1.027 times. With the SAT solver X-CSE solves 99 and I-CSE-Iter solves 99 instances, and the geometric mean speed-up of X-CSE is 1.360 times.

I-CSE-Iter introduces additional variables and constraints so there is a risk of slowing down the solver. On the Car Sequencing problem, both the CP and SAT solvers are slowed down while the CP search tree is unchanged.

Comparing I-CSE-Iter to I-CSE, the geometric mean speed-up of total time with the CP solver is 1.010 (with confidence interval [0.982, 1.045]) and with the SAT solver

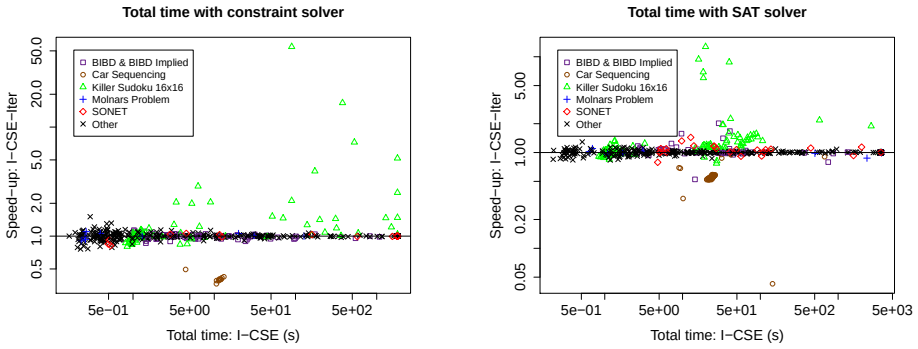

Figure 5: Comparing I-CSE-Iter to I-CSE (total time) with the CP solver (left) and the SAT solver (right).

is 0.972 (with confidence interval [0.946, 1.001]). On average it seems there is little difference between I-CSE-Iter and I-CSE.

2.5. Experiment E: Active Associative-Commutative CSE

Active X-CSE extends X-CSE on sums by negating coefficients. Apart from sums, the two configurations are identical. Figure 6 compares the two configurations. Active AC-CSs are found in problems Tic Tac Toe, Plotting and Nurse Rostering, however the instances may not be hard enough to show any improvement. All instances of these three classes are solved in less than three seconds and fewer than 63000 nodes with the CP solver. Active X-CSE does not change the search tree (with the CP solver) for any instance that did not time out. The Active X-CSE algorithm takes substantially longer than X-CSE on seven instances of Car Sequencing while the CP solver takes the same time (Figure 6 left).

With the SAT solver Car Sequencing is also slowed during tailoring (with no effect on the solver). Instances of BIBD and SONET are scattered. The two algorithms have the same heuristic however when the heuristic ties they may extract (standard) sum AC-CSs in different orders and thus produce a different final model, and this appears to be affecting the SAT solver's search heuristic. BIBD appears to slow down on average and it is not clear why this is the case.

The geometric mean speed-up of total time with the CP solver is 0.980 (with confidence interval [0.966, 0.992]) and with the SAT solver is 0.893 (with confidence interval [0.873, 0.914]).

References

[1] Araya, I., Neveu, B., Trombettoni, G.: Exploiting common subexpressions in numerical CSPs. In: Principles and Practice of Constraint Programming (CP 2008). pp. 342–357 (2008)

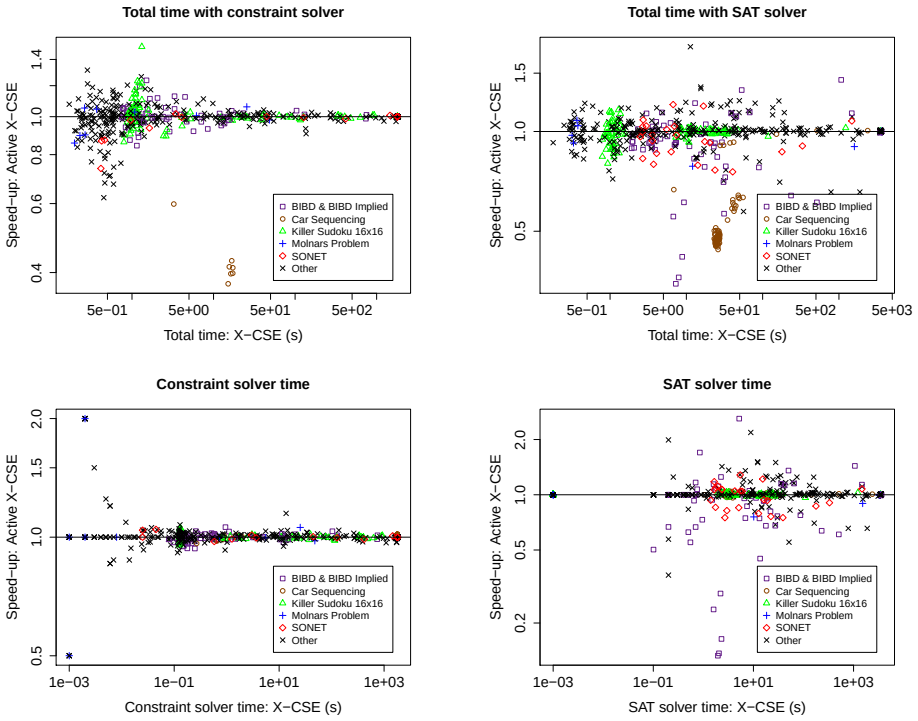

Figure 6: Comparing Active X-CSE to X-CSE with the CP solver (left) and the SAT solver (right), total time (above) and solver time only (below).

[2] Bacchus, F.: GAC via unit propagation. In: Principles and Practice of Constraint Programming (CP 2007). pp. 133–147 (2007)

[3] Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Global constraints for lexicographic orderings. In: van Hentenryck, P. (ed.) Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming. pp. 93–108 (2002)

[4] Gecode Team: Gecode: Generic constraint development environment (2016), available from `http://www.gecode.org`

[5] Gent, I.P.: Arc consistency in SAT. In: Proceedings of the 15th European Conference on Artificial Intelligence (ECAI 2002). pp. 121–125 (2002)

[6] Gent, I.P., Nightingale, P.: A new encoding of AllDifferent into SAT. In: Proc. 3rd International Workshop on Modelling and Reformulating Constraint Satisfaction Problems (ModRef 2004). pp. 95–110 (2004)

[7] Leo, K., Tack, G.: Multi-pass high-level presolving. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI). pp. 346–352 (2015)

[8] Nightingale, P., Özgür Akgün, Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Submitted to Artificial Intelligence (2017), under review

[9] Tamura, N., Taga, A., Kitagawa, S., Banbara, M.: Compiling finite linear CSP into SAT. Constraints 14(2), 254–272 (2009)