Automatic Generation and Selection of Streamlined Constraint Models via Monte Carlo Search on a Model Lattice

Patrick Spracklen, Özgür Akgün, Ian Miguel

School of Computer Science, University of St Andrews, St Andrews, UK
{jlps, ozgur.akgun, ijm}@st-andrews.ac.uk

Abstract. Streamlined constraint reasoning is the addition of uninferred constraints to a constraint model to reduce the search space, while retaining at least one solution. Previously it has been established that it is possible to generate streamliners automatically from abstract constraint specifications in Essence and that effective combinations of streamliners can allow instances of much larger scale to be solved. A shortcoming of the previous approach was the crude exploration of the power set of all combinations using depth and breadth first search. We present a new approach based on Monte Carlo search over the lattice of streamlined models, which efficiently identifies effective streamliner combinations.

1 Introduction and Background

If the performance of a constraint model is found to be inadequate, a natural step is to consider adding constraints to the model in order to assist the constraint solver in detecting dead ends in search and therefore reducing overall search effort. One approach is to add *implied* constraints, which can be inferred from the initial model and are therefore guaranteed to be sound. Effective implied constraints have been found both by hand [16, 17] and via automated methods [9, 10, 18]. If implied constraints cannot be found, or improve performance insufficiently, for satisfiable problems[1] a more aggressive step is to add *streamliner* constraints [21], which are not guaranteed to be sound but are designed to reduce significantly the search space while permitting at least one solution. For several problem classes, effective streamliners have been found by hand by looking for patterns in the solutions of small instances of those classes [21, 23, 25, 26].

Wetter et al. [39] demonstrated how to generate effective streamliners automatically from the specification of a constraint problem class in the abstract constraint specification language Essence [13–15]. This method, which we expand upon, exploits the structure apparent in an Essence specification, such as that of the Progressive Party Problem (Fig. 1), to produce candidate streamliners via a set of streamliner generation rules. An effective streamliner that we

[1] Streamlining is unsuitable for unsatisfiable problems: streamliners are not necessarily sound, so exhausting the search space does not prove unsatisfiability (a case split approach is possible: a sub-problem with a streamliner, another with its negation).

```
1  language Essence 1.3
2  given n_boats, n_periods : int(1..)
3  letting Boat be domain int(1..n_boats)
4
5  find hosts : set (minSize 1) of Boat,
6    sched : set (size n_periods) of function (total) Boat --> Boat
7  minimising |hosts|
8
9  such that
10 $ Hosts remain the same throughout the schedule
11 forAll p in sched . range(p) = hosts,
12 $ Hosts stay on their own boat
13 forAll p in sched . forAll h in hosts . p(h) = h,
14 $ Hosts have the capacity to support the visiting crews
15 forAll p in sched . forAll h in hosts .
16   (sum b in preImage(p,h) . crew(b)) <= capacity(h),
17 $ No two crews are at the same party more than once
18 forAll b1,b2 : Boat .
19   b1 < b2 -> (sum p in sched . toInt(p(b1) = p(b2))) <= 1
```

Fig. 1. The Progressive Party Problem [35] in Essence. There are two abstract decision variables, a set of host boats, and a set of functions from boats to boats representing the assignment of guests to hosts in each period. From this very concise, structured statement of the problem, 160 candidate streamliners can be generated by our system.

automatically generate for this problem class constrains approximately half of the entries in the `sched` set variable to be monotonically increasing functions.

Using training instances drawn from the problem class under consideration, streamliner candidates are evaluated via a toolchain consisting of the automated constraint modelling tools Conjure [1–4] and Savile Row [30–32], and the constraint solver Minion [20]. Promising candidates, which retain at least one solution to the training instances while significantly reducing search, are used to solve more difficult instances from the same problem class. Candidate streamliners are often most effectively used in combination. For example, Wetter et al. presented an effective combination of three streamliners for the Van der Waerden numbers problem. Hence, the space of streamlined models forms a lattice where the root is the original Essence specification and an edge represents the addition of a streamliner to the combination associated with the parent node.

A shortcoming of Wetter et al.'s work is the uninformed manner in which the streamliner lattice is explored using depth- or breadth-first search. Our principal contribution is a new method for exploring the lattice via Monte Carlo-style search, allowing more effective streamlined models to be found in a time budget. A second contribution is a set of new streamliner generator rules for sequence and matrix Essence type constructors to complement those presented by Wetter et al. Finally, we demonstrate the efficacy of our approach on a variety of problems.

Name	matrix all
Param	`R` (another rule)
Input	`X: matrix indexed by [I] of _`
Output	`forAll i : I . R(X[i])`

Name	matrix most (similarly for matrix half and matrix approximately half)
Param	`R` (another rule)
Input	`X: matrix indexed by [I] of _`
Output	`|I| / 2 <= sum i : I . toInt(R(X[i]))`

Name	sequence monotonically increasing (similarly for monotonically decreasing)
Input	`X: sequence of _`
Output	`forAll i,j in defined(X). i < j -> X(i)<= X(j)`

Name	sequence smallest first (similarly for largest first)
Input	`X: sequence of _`
Output	`forAll i in defined(X). X(min(defined(X)))<= X(i)`

Name	sequence on defined (similarly for range)
Param	`R` (another rule)
Input	`X: sequence of _`
Output	`R(defined(X))`

Fig. 2. Streamliner generators for sequence and matrix domains.

2 Essence Specifications and Streamliner Generators

An Essence specification such as that presented in Fig. 1 identifies: the input parameters of the problem class (`given`), whose values define an instance; the combinatorial objects to be found (`find`); the constraints the objects must satisfy (`such that`); identifiers declared (`letting`); and an (optional) objective function (`min/maximising`). The key feature of the language is its support for abstract decision variables, such as set, multiset, relation and function, as well as *nested* types, such as the set of functions found in Fig. 1.

A concise, structured specification of a problem class directly supports the generation of powerful candidate streamliners: in the example, it is readily apparent that the problem requires us to find a set of boats and a set of functions assigning guest boat crews to hosts. Hence, streamliners related to sets and functions, such as that given in the introduction, can be generated straightforwardly. In contrast an equivalent constraint model in, for example, MiniZinc [29] or Essence Prime [31] has to represent these abstract decision variables with constrained collections of more primitive (e.g. integer domain) variables, such as the matrix model [11, 12] proposed by Smith et al. [35]. In this context, it is significantly more difficult to recognise the structure (i.e. the set and set of functions) in the problem and generate the equivalent streamliners.

Wetter et al. present a set of streamliner generation rules for the Essence type constructors set, function and partition, as well as simple integer domains [39]. Our first contribution is to extend this set also to cover sequence and matrix

type constructors. These are summarised in Figure 2. For decision variables with matrix domains, one generator (matrix all) takes another streamliner generator as a parameter (`R`, as a simple example: constrain an integer variable to take an even value) and *lifts* it to operate on all entries in a matrix. This rule can be applied to higher dimensional matrix domains as well, in which case the multi-dimensional matrix domain is interpreted in the same way as a series of nested one-dimensional matrix domains. The generators matrix most (and matrix half and matrix approximately half) operate in a similar way. In contrast to the matrix all streamliner generator, these generators first reify the result of applying `R`, and then restrict the number of places the constraint must hold.

For sequence domains, we present two sets of first-order streamliner generators: monotonically increasing(or decreasing) and smallest (or largest) first. These do not take another generator as a parameter but directly post constraints on the sequence decision variable. The sequence on range and sequence on defined generators take an existing streamliner generator as a parameter and lift it to work on the range or the defined set of the sequence domain respectively.

3 Monte Carlo Search for Streamliner Combinations

Le Bras et al. [24] and Wetter et al. [39] both observed that by applying several streamlining constraints to a model simultaneously the search required for finding the first solution can be reduced further than by applying the streamliners in isolation. Finding an effective streamliner combination involves searching the streamliner lattice, the size of which is determined by the power set of all candidate streamliners for a given problem class. Table 1 presents the number of candidate streamliners our current set of generation rules produces for a number of problem classes. In some cases the number of candidates generated is small. However, the cost of evaluating each combination on a set of test instances means that it is typically not feasible to evaluate all possible streamliner combinations.

Wetter et al. employed breadth-first and depth-first search to explore the streamliner lattice in an uninformed manner. The motivation for our work is the hypothesis that a best-first search can allow more effective streamliner combinations to be discovered within a given time budget. Our approach is to focus the search onto areas of the lattice where the streamliners combine to give the greatest reduction in search while retaining at least one solution.

For a given problem class, we have no prior knowledge of the performance of the set of candidate streamliners, either individually or in combination. This raises the issue of the exploration/exploitation problem: if we can identify a combination of streamliners that performs well, should we try and exploit that combination further by evaluating the addition of further streamliners, or should we explore other parts of the lattice that may at present seem less promising?

The exploration/exploitation tradeoff can be formalised in several reinforcement learning variants, including via markov decision processes [**?**]. We model this situation as a multi-armed bandit problem [5], which allows us to employ regret minimising algorithms to deal with the exploration/exploitation dilemma.

The multi-armed bandit can be seen as a set of real distributions, each distribution being associated with the rewards delivered by one of the K levers. Since the multi-armed bandit problem assumes that each lever corresponds to an independent action, in order to use it directly we would have to associate a lever with each point in the streamliner lattice, which is infeasible in general. Instead, we use the bandit algorithm to guide the exploration of the lattice in a process reminiscent of Monte Carlo Tree Search (MCTS) [7], as described below.

3.1 Algorithm Outline

Our algorithm has the same four basic steps as MCTS. It uses Upper Confidence bound applied to Trees (UCT) [7] to balance exploration and exploitation.

1. **Selection**: Starting at the root node, the UCT policy is recursively applied to traverse the explored part of the lattice until an unexpanded, but expandable node is reached. A node is expandable if it has at least one child that is not marked as pruned (§3.5). A child node is selected to maximise:

$$UCT = X_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

   where n is the number of times the current (parent) node in the lattice has been visited, n_j the number of times child j has been visited, X_j is the mean reward associated with child j and $C_p > 0$ is a constant [7].
2. **Expansion**: Enumerate the children of the Selected node and choose one to expand according to the heuristic explained in Section 3.4.
3. **Simulation**: The collection of streamliners associated with the expanded node are evaluated using the Conjure, Savile Row, and Minion toolchain.
4. **BackPropagation**: The result of the evaluation is propagated back up through the lattice to update reward values, as explained below.

3.2 Back Propagation

Since our search is operating over a lattice, a node may have multiple parents. This requires an alteration to the back propagation employed in MCTS: when we perform back propagation that reward value is back propagated up all paths from that node to the root. To illustrate consider a problem with two streamliners {A,B} and we are back propagating from a node in the lattice representing the combination {AB}. There are two paths by which this node could have been reached, {root $\rightarrow$ A $\rightarrow$ B} and {root $\rightarrow$ B $\rightarrow$ A}. Although the algorithm will have only descended one of these paths, because the reward value of a node in the lattice is representative of the ability of the streamliner combination represented by that node to combine and produce effective reductions in search, the node in the lattice that represents streamliner combination {B} should also receive this reward. For this reason both paths are rewarded accordingly and the reward generated is back propagated up all paths from that node to the root. We also

ensure that a node that lies on more than one such path is rewarded only once. The cost of back propagation thus grows exponentially with depth. However, since each level of the lattice represents an additional constraint it is unlikely that satisfiability is maintained at great enough depths for this to become an issue. Empirically, the cost is insignificant relative to solving the training instances.

We must also consider the situation where a node in a path back to the root has not yet been expanded. If we ignore such nodes, their true reward is not reflected in their reward values because all reward values back propagated from child nodes prior to their creation are lost. Our approach is that when a node is expanded, it absorbs the reward value and visit count of its immediate children that have already been expanded. This avoids caching a potentially large set of values while maintaining reward values for nodes around the focus of our search.

3.3 Simulation Reward

The performance of our best first search algorithm is heavily reliant on how the reward is produced from the simulation step. Initially we assigned rewards as follows: if the majority of the instances evaluated are unsatisfiable a reward of -1 is back-propagated, otherwise a reward of one minus the average reduction in search space (expressed in search nodes) is returned. While this is valid, our initial experiments revealed that its effect was to produce a search strategy similar to breadth-first search - i.e. too strong a bias towards exploration.

The reason for this is that the penalty value is too punitive when evaluating larger streamliner combinations. Intuitively, the penalty should be sensitive to the depth we have travelled into the lattice: as we add streamliners we reduce the search space and we expect the probability of such failure to rise. Therefore we divide the penalty value by the depth of the node being evaluated, allowing the prolonged exploration of promising paths.

3.4 Expansion Heuristic

The order of expansion of child nodes is an important factor in performance. An expanded child consumes time to perform simulation and, because the simulation reward is back propagated, if a penalty is awarded it can affect the likelihood of the parent node being selected on the next iteration. During the expansion phase of our algorithm child nodes are expanded in descending likelihood of the application of the associated streamliner combination resulting in a satisfiable problem. In order to facilitate this, when a successful simulation is performed, for a representative instance the solution found is stored, along with the approximate size of its search space (via the product of the domains of the decision variables in the model) and the proportion of the space explored to find the solution.

Upon expansion all potential children are enumerated and for each we check whether the additional associated streamliner invalidates the solution stored at the expanding parent. If the solution remains valid then the child is preferred for expansion because we know pre-simulation that the associated streamliner combination satisfies at least one instance and the additional streamliner might

reduce search. If the solution is invalidated then the search space explored by the child is smaller than the expanding parent. We use the proportion of search space explored to find the solution associated with the expanding parent to estimate the likelihood of a solution existing in that subspace. Intuitively, if the parent explored a large fraction of the space then it is less likely that a solution will be found when adding the streamliner associated with the child node.

3.5 Pruning the Streamliner Lattice

As per §3.2, when a simulation for a streamliner combination reveals that the majority of training instances are unsatisfiable, a penalty is back propagated up the lattice. We also mark the node associated with the simulation as pruned and never consider any of its children for expansion. In addition, we prune nodes whose additional streamliner is determined to be *redundant* in combination with those inherited from the expanding parent, in the sense that it causes no further reduction in search on the evaluated instances. Pruning the lattice by these two methods typically reduces the number of nodes to be expanded very significantly.

4 Empirical Evaluation

We evaluate two hypotheses empirically. First, that the best-first search is more effective in exploring the streamliner lattice than the simpler depth- and breadth-first search methods employed in [39]. Second, that our method is able to automatically find streamliner combinations that drastically reduce the search space across a variety of problem classes.

We experiment with thirteen problem classes, eight from Wetter et al. and the remainder selected for variety, particularly problem classes requiring significantly more instance data such as SONET [27]. Streamlining can aid in the search for feasible solutions of optimisation problems, but not the proof of optimality. Hence, in our experiments we transformed optimisation into decision problems by the standard method of bounding the objective and searching for a satisfying solution. The results we obtain are very positive, as presented in Table 1.

5 Conclusion

We have presented a new method for the automated generation of streamlined constraint models from a large set of candidates via Monte Carlo search. Our method is efficient in searching the space of candidates, producing more effective streamlined models in less time than less informed approaches. Our empirical results demonstrate a vast reduction in search across a variety of benchmarks.

As part of future work we plan to explore the generation of streamlined versions of alternative models generated by Conjure. We expect the utility of particular streamlining constraints to vary depending on the model.

Acknowledgements This work was supported via EPSRC EP/P015638/1. We thank our anonymous reviewers for helpful comments.

Problem Class	Number of Candidate Streamliners	Mean Instance Time (Seconds)	DFS		BFS		Monte Carlo		
			Combination size	Mean Reduction in Search Nodes	Combination size	Mean Reduction in Search Nodes	Combination size	Mean Reduction in Search Nodes	Mean Reduction in Search Time
Progressive Party [35]	160	76	1	48.2%	2	41.5%	4	93.2%	91.7%
MEB [8]	76	54	2	72.7%	2	83.1%	3	96.2%	93.5%
Schur's Lemma [38]	65	254	1	39.2%	1	81.4%	3	92.3%	89.8%
Quasigroup Existence [34]	17	337	1	83.0%	1	83.0%	1	83.0%	87.4%
Van Der Waerden Numbers [37]	65	142	3	84.4%	2	82.3%	4	96.2%	95.7%
Graceful Double Wheel Graphs [26]	72	572	2	65.6%	3	81.0%	4	94.2%	90.2%
Graceful Gears [28]	72	493	2	64.6%	2	84.9%	4	95.5%	89.1%
Graceful Helms [6]	72	598	2	85.4%	2	80.6%	3	91.8%	87.1%
Graceful Wheel Graphs [19]	72	12	2	69.5%	2	67.4%	4	88.3%	80.8%
Car Sequencing [33]	36	724	1	31.9%	1	31.9%	1	31.9%	37.9%
EFPA [22]	144	231	1	86.1%	1	86.1%	1	86.1%	85.7%
SONET [27]	64	678	1	72.0%	2	87.2%	3	94.6%	95.8%
CVRP [36]	84	817	1	93.0%	1	93.0%	1	93.0%	84.1%

Table 1. For each of the thirteen problem classes, fifteen instances were split 70/30 into training and test sets. All three methods received the same training budget of six hours per problem class, a cost which is amortised over the entire problem class for which the streamliners are applicable. We record the number of candidate streamliners and the mean time to solution for the test instances using non-streamlined models in columns 2 and 3. For the substantial majority of the problem classes the most effective streamliner discovered is composed of a combination of individual streamlining constraints, as presented in columns 4, 6 and 8. In these cases the Monte Carlo search method is able to discover larger combinations that yield superior results. For the problem classes where a single streamliner was found to be the most effective all methods are equally effective, as through pruning (§3.5) we were able to exhaustively search the space of all streamliner combinations. Mean reduction for both time and search nodes is measured by comparing the search required to find the first solution on the non-streamlined model with the model streamlined with the most effective streamliner combination found using the respective search method. The selected streamliners for each problem class retained satisfiability on all test instances. Through the addition of streamlining constraints we obtain a uniformly vast reduction in both search nodes and time. All computational experiments were run on a 32-core AMD Opteron 6272 at 2.1 GHz and an 8 core Intel Core i7-6920HQ at 2.90GHz. All experiments for an individual problem class were run on a single machine.
Experimental data, models, raw results, and source code can be downloaded from: http://github.com/stacs-cp/cp2018-streamlining

References

1. Akgün, Ö.: Extensible automated constraint modelling via refinement of abstract problem specifications. Ph.D. thesis, University of St Andrews (2014)
2. Akgün, Ö., Frisch, A.M., Gent, I.P., Hussain, B.S., Jefferson, C., Kotthoff, L., Miguel, I., Nightingale, P.: Automated symmetry breaking and model selection in Conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 107–116. Springer (2013)
3. Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P.: Breaking conditional symmetry in automated constraint modelling with Conjure. In: ECAI. pp. 3–8 (2014)
4. Akgün, Ö., Miguel, I., Jefferson, C., Frisch, A.M., Hnich, B.: Extensible automated constraint modelling. In: Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence. pp. 4–11. AAAI Press (2011)
5. Auer, P., Cesa-Bianchi, N., Fischer, P.: Finite-time analysis of the multiarmed bandit problem. Machine Learning **47**(2), 235–256 (May 2002). https://doi.org/10.1023/A:1013689704352
6. Ayel, J., Favaron, O.: Helms are graceful. Progress in Graph Theory (Waterloo, Ont., 1982), Academic Press, Toronto, Ont pp. 89–92 (1984)
7. Browne, C., Powley, E., Whitehouse, D., Lucas, S., Cowling, P.I., Tavener, S., Perez, D., Samothrakis, S., Colton, S., et al.: A survey of monte carlo tree search methods. IEEE Transactions on Computational Intelligence and AI (2012)
8. Čagalj, M., Hubaux, J.P., Enz, C.: Minimum-energy broadcast in all-wireless networks: Np-completeness and distribution issues. In: Proceedings of the 8th annual international conference on Mobile computing and networking. pp. 172–182. ACM (2002)
9. Charnley, J., Colton, S., Miguel, I.: Automatic generation of implied constraints. In: ECAI. vol. 141, pp. 73–77 (2006)
10. Colton, S., Miguel, I.: Constraint generation via automated theory formation. In: International Conference on Principles and Practice of Constraint Programming. pp. 575–579. Springer (2001)
11. Flener, P., Frisch, A., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling. In: Proc. of the CP-01 Workshop on Modelling and Problem Formulation. p. 223 (2001)
12. Flener, P., Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Matrix modelling: Exploiting common patterns in constraint programming. In: Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems. pp. 27–41 (2002)
13. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The essence of essence. Modelling and Reformulating Constraint Satisfaction Problems p. 73 (2005)
14. Frisch, A.M., Grum, M., Jefferson, C., Hernández, B.M., Miguel, I.: The design of essence: A constraint language for specifying combinatorial problems. In: IJCAI. vol. 7, pp. 80–87 (2007)
15. Frisch, A.M., Harvey, W., Jefferson, C., Martínez-Hernández, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13**(3), 268–306 (2008)
16. Frisch, A.M., Jefferson, C., Miguel, I.: Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In: ECAI. vol. 16, p. 171 (2004)

17. Frisch, A.M., Miguel, I., Walsh, T.: Symmetry and implied constraints in the steel mill slab design problem. In: Proc. CP01 Workshop on Modelling and Problem Formulation (2001)
18. Frisch, A.M., Miguel, I., Walsh, T.: Cgrass: A system for transforming constraint satisfaction problems. In: Recent Advances in Constraints, pp. 15–30. Springer (2003)
19. Frucht, R.: Graceful numbering of wheels and related graphs. Annals of the New York Academy of Sciences **319**(1), 219–229 (1979)
20. Gent, I.P., Jefferson, C., Miguel, I.: Minion: A fast scalable constraint solver. In: ECAI. vol. 141, pp. 98–102 (2006)
21. Gomes, C., Sellmann, M.: Streamlined constraint reasoning. In: International Conference on Principles and Practice of Constraint Programming. pp. 274–289. Springer (2004)
22. Huczynska, S., McKay, P., Miguel, I., Nightingale, P.: Modelling equidistant frequency permutation arrays: An application of constraints to mathematics. In: International Conference on Principles and Practice of Constraint Programming. pp. 50–64. Springer (2009)
23. Kouril, M., Franco, J.: Resolution tunnels for improved sat solver performance. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 143–157. Springer (2005)
24. Le Bras, R., Gomes, C.P., Selman, B.: Double-wheel graphs are graceful. In: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence. pp. 587–593. IJCAI '13, AAAI Press (2013), http://dl.acm.org/citation.cfm?id=2540128.2540214
25. Le Bras, R., Gomes, C.P., Selman, B.: On the erdős discrepancy problem. In: International Conference on Principles and Practice of Constraint Programming. pp. 440–448. Springer (2014)
26. LeBras, R., Gomes, C.P., Selman, B.: Double-wheel graphs are graceful. In: IJCAI. pp. 587–593 (2013)
27. Lee, Y., Sherali, H.D., Han, J., Kim, S.i.: A branch-and-cut algorithm for solving an intraring synchronous optical network design problem. Networks **35**(3), 223–232 (2000)
28. Ma, K., Feng, C.: On the gracefulness of gear graphs. Math. Practice Theory **4**, 72–73 (1984)
29. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: International Conference on Principles and Practice of Constraint Programming. pp. 529–543. Springer (2007)
30. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I.: Automatically improving constraint models in Savile Row through associative-commutative common subexpression elimination. In: 20th International Conference on Principles and Practice of Constraint Programming (CP 2014). pp. 590–605. Springer (2014)
31. Nightingale, P., Akgün, O., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001
32. Nightingale, P., Spracklen, P., Miguel, I.: Automatically improving SAT encoding of constraint problems through common subexpression elimination in Savile Row. In: Proceedings of the 21st International Conference on Principles and Practice of Constraint Programming (CP 2015). pp. 330–340. Springer (2015)
33. Parrello, B.D., Kabat, W.C., Wos, L.: Job-shop scheduling using automated reasoning: A case study of the car-sequencing problem. Journal of Automated reasoning **2**(1), 1–42 (1986)

34. Slaney, J., Fujita, M., Stickel, M.: Automated reasoning and exhaustive search: Quasigroup existence problems. Computers & mathematics with applications **29**(2), 115–132 (1995)
35. Smith, B.M., Brailsford, S.C., Hubbard, P.M., Williams, H.P.: The progressive party problem: Integer linear programming and constraint programming compared. Constraints **1**(1-2), 119–138 (1996)
36. Toth, P., Vigo, D.: The vehicle routing problem. SIAM (2002)
37. van der Waerden, B.: Beweis einer Baudetschen Vermutung. Nieuw Arch. Wisk. **19**, 212–216 (1927)
38. Walsh, T.: CSPLib problem 015: Schur's lemma . http://www.csplib.org/Problems/prob015
39. Wetter, J., Akgün, Ö., Miguel, I.: Automatically generating streamlined constraint models with Essence and Conjure. In: International Conference on Principles and Practice of Constraint Programming. pp. 480–496. Springer (2015)