Automatically Transforming Constraint Satisfaction Problems: Further Progress

Alan M. Frisch, Ian Miguel
AI Group
Dept. Computer Science
University of York, York, England
{frisch,ianm}@cs.york.ac.uk

Toby Walsh
Cork Constraint Computation Center
University College Cork
Ireland
tw@4c.ucc.ie

1 Introduction

A constraint satisfaction problem (CSP) consists of a triple, (X, D, C), where X is a set of variables, D is a set of corresponding domains of values and C is a set of constraints which specify allowed combinations of assignments of values to variables. Constructing an effective model of a CSP is, however, a challenging task. Carefully chosen transformations of a basic model can greatly reduce the amount of effort required to solve the problem by systematic search (e.g. [5]). These include adding constraints that are implied by other constraints in the problem, adding constraints that remove symmetrical solutions to the problem, removing redundant constraints and replacing constraints with their logical equivalents.

1.1 CGRASS

Outwith a highly focused domain like planning [2], model transformation is typically performed in an ad-hoc manner by hand. We describe the CGRASS (Constraint GeneRation And Symmetry-breaking) system that can improve a problem model by automatically performing appropriate transformations. The system architecture is based on, and extends, Bundy's 'proof planning' [1].

Proof planning is used to guide the search for a proof in automated theorem proving. Common patterns in proofs are identified and encapsulated in *methods*. A proof planner takes a goal to prove, and selects from a database of methods one which matches this goal. The proof planner checks that the pre-conditions of the method hold. If so the proof planner executes the post-conditions. This constructs the output goal or goals.

Strong method preconditions limit transformations to those that are likely to produce a simplified problem. Methods act at a high level, performing complex transformations that might require complex proofs to justify at the individual inference rule level. Search control is cleanly separated from the inference steps. We can therefore try out a variety of search strategies.

CGRASS' input currently consists of a statement of the initial problem variables and their domains and a list of constraints. We intend to enable CGRASS to accept the OPL language [6] as input, as well as specialised extensions such as the ability to support matrices of variables directly. Output is created in the same simple language. Hence, very little effort is necessary for translation into the required input for a variety of existing solvers.

Fundamental to the efficiency of CGRASS is the normal form it imposes on the constraint set, inspired by that used in the HARTMATH computer algebra system[1]. A normal form allows us, to an extent, to replace the test for semantic equivalence with a simpler syntactic comparison. The normal form used combines a lexicographic ordering and simplifications such as collection of like terms, cancellation and removal of common factors.

2 Methods

The following set of methods is not complete in any sense, and a principle item of future work is its extension.

Symmetry Often the most useful constraints can only be derived when some or all symmetry has been broken. Hence, CGRASS attempts to detect symmetry as a pre-processing step. Symmetrical variables have identical domains and are such that, if all occurrences of the pair in the constraint set are exchanged and the constraint set is re-normalised it returns to its original state. A similar process is used to check for symmetrical sub-terms. Symmetry is broken by creating a partial order between symmetrical variables/sub-terms.

Introduce This method binds a new variable to a non-atomic sub-term that recurs in the constraint set. Variable introduction is a powerful tool which, in combination with `eliminate` (below), can tighten the constraint graph and reduce constraint arity. Tightening the constraint graph means that propagation on the introduced variable's domain has a wider reaching effect. Reducing a constraint's arity means that fewer variables need to be instantiated before it can be used to prune the search space.

Eliminate This method performs Gaussian-like variable elimination. The preconditions identify variables or terms which can be eliminated from a more complex constraint, insisting that the resulting constraint has a smaller size (in terms of number of constituent terms) than the original.

[1] `http://www.hartmath.org`

Other Methods The `genAllDiff` method greedily generates an all-different constraint (for which we have powerful pruning methods [4]) from a clique of inequations. Other simple, but useful, methods also exist. For example, `nodeConsistency` prunes domain elements given an algebraic constraint consisting of a variable and a constant. These methods are not only cheap to fire, but often result in a reduction in the size of the constraint set. This promotes efficiency by leaving fewer constraints for the more complicated methods to attempt to match against.

3 Issues

In constructing CGRASS we have extended proof planning along a number of dimensions:

Non-monotonicity: CGRASS' methods sometimes add a new constraint, at others replace a constraint by a tighter one, or eliminate a redundant constraint. The constraint set may therefore increase or decrease. Hence, we replace the method 'output' by 'add' and 'delete' lists as used in classical planning.

Pattern matching: Existing proof planners typically use first order unification to match a method's input against the current goal or subgoal. CGRASS uses a richer pattern matching language specialised to the task of reasoning about sets of constraints.

Looping: Unless a method deletes some of its input constraints, its preconditions continue to hold, allowing repeated firing. We incorporated an history mechanism into CGRASS to prevent this.

Termination: Previous applications of proof planning have a clear termination condition: goals are reduced to subgoals until all are proven. In transforming CSPs it is much less clear. We must decide when to stop making transformations and search for an answer. CGRASS' methods currently have strong enough preconditions that they can be run to exhaustion. We may in the future have to add a proof critic-style executive [3] which terminates proof planning when future rewards look poor.

Constraint utility: CGRASS uses measures like constraint arity and tightness to eliminate obviously useless constraints. We are inventing heuristics to help with the difficult decision as to which of the remaining derived constraints to keep.

Explanation: In order for the user to see how a new model was derived, we adapted the existing proof planning tactic mechanism. CGRASS' tactics write out text explaining the application of the methods.

4 The Golomb Ruler Problem

We illustrate the operation of CGRASS by means of the Golomb ruler problem [5]. A Golomb ruler is a set of n ticks at integer points on a ruler of length m such that all the inter-tick distances are unique. A concise model:

$$\text{minimise:}\max_i(x_i)$$

$$\forall i,j,k,l\ (i \neq j) \wedge (k \neq l) \wedge (i \neq k \vee j \neq l) \rightarrow$$
$$(x_i - x_j \neq x_k - x_l) \text{ is added to the problem}$$

Taken literally, this a poor model. The constraints are quaternary, and will be delayed by most solvers. There is also a large amount of symmetry present.

We focus on the 3-tick ruler. The basic model produces 30 constraints. CGRASS' initial normalisation of the constraint set reduces this to just 12. Symmetry testing reveals that x_1, x_2 and x_3 are symmetrical, hence $x_1 \leq x_2$ and $x_2 \leq x_3$ are added. Three variables, $z_0 = x_1 - x_2$, $z_1 = x_2 - x_3$ and $z_2 = x_3 - x_1$ are `introduced` and `eliminate` substitutes them into the quaternary constraints. Finally, `genAllDiff` fires which, coupled with the firing of some of the minor auxiliary methods, gives the concise model of table 1.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_1 = x_2 - x_3$	$z_2 = x_3 - x_1$
All-different(z_0, z_1, z_2)		

Table 1: Golomb Ruler: Transformed Model.

This model is far easier to solve than the original. Similar models are obtained for larger Golomb rulers, leading to substantial reductions in search effort. However, larger problems generate a greater volume of input constraints via the naive input model. This creates a lot of work for CGRASS and an overall reduction in efficiency.

5 Next Steps

In order to compensate, we are extending CGRASS to support quantified expressions directly. This would immediately reduce the size of the input to two constraints in the Golomb ruler example. It would also allow us to reason about a whole *class* of problems, rather than at the level of single instances. Writing methods would typically be complicated by the need to support quantified constraints. However, certain operations would be made much easier. One example is the `genAllDiff` method where the input would be simply: $\forall i,j\ \ i \neq j \rightarrow x_i \neq x_j$.

Acknowledgements

This project and the second author are supported by EPSRC Grant GR/N16129[2].

References

[1] A. Bundy. A science of reasoning. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press, 1991.

[2] M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proc. 15th IJCAI*, pages 1169–1176, 1997.

[3] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proof. In *Proceedings of LPAR'92, LNAI 624*. Springer-Verlag, 1992.

[4] J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. 12th AAAI*, pages 362–367, 1994.

[5] B.M. Smith, K. Stergiou, and T. Walsh. Modelling the golomb ruler problem. In *Proceedings of the IJCAI-99 Workshop on Non-Binary Constraints*. International Joint Conference on Artificial Intelligence, 1999.

[6] P. van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

[2] `http://www.cs.york.ac.uk/aig/projects/implied/index.html`