CGRASS: A System for Transforming Constraint Satisfaction Problems

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

Abstract. Experts at modelling constraint satisfaction problems (CSPs) carefully choose model transformations to reduce greatly the amount of effort that is required to solve a problem by systematic search. It is a considerable challenge to automate such transformations and to identify which transformations are useful. Transformations include adding constraints that are implied by other constraints, adding constraints that eliminate symmetrical solutions, removing redundant constraints and replacing constraints with their logical equivalents. This paper describes the CGRASS (Constraint Generation And Symmetry-breaking) system that can improve a problem model by automatically performing transformations of these kinds. We focus here on transforming individual CSP instances. Experiments on the Golumb Ruler problem suggest that producing good problem formulations solely by transforming problem instances is, generally, infeasible. We argue that, in certain cases, it is better to transform the problem class than individual instances and, furthermore, it can sometimes be better to transform formulations of a problem that are more abstract than a CSP.

1 Introduction

Constraint satisfaction is a successful technology for tackling a wide variety of search problems including resource allocation, transportation and scheduling. Constructing an effective model of a constraint satisfaction problem (CSP) is, however, a challenging task as new users typically lack specialized expertise. One difficulty is in identifying transformations, which are sometimes complex, that can dramatically reduce the effort needed to solve the problem by systematic search (see, for example, [9]). Such transformations include adding constraints that are implied by other constraints in the problem, adding constraints that eliminate symmetrical solutions to the problem, removing redundant constraints and replacing constraints with their logical equivalents. Unfortunately, outside a highly focused domain like planning (see, for example, [3]), there has been little research on how to perform such transformations automatically.

Our initial focus is on the utility of transforming individual CSP instances. We show how Bundy's *proof planning* [1] technology can be extended and adapted to this task. The CGRASS system (Constraint Generation And Symmetry-breaking) is described and its operation illustrated on the Golomb ruler problem [9], a difficult combinatorial problem with many applications. Our results suggest that making transformations to single problem instances alone is not generally practical. We argue that it is essential to make transformations at the level of abstraction at which they are most straightforward. For example, we might also consider quantified constraint expressions and high level descriptions of an entire problem *class*. CGRASS provides an extensible platform which will, in future, support transformations at multiple levels of abstraction.

2 Proof Planning

Proof planning is a technique used to guide the search for a proof in automated theorem proving. Common patterns in proofs are identified and encapsulated in *methods*. Proof planning has often been associated with “rippling”, a powerful heuristic for guiding search in inductive proof. However, proof planning can easily be adapted to other mathematical tasks like finding closed form sums to series or, as here, transforming constraint satisfaction problems.

A proof planner like CLAM [2] 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 (which are a sequence of statements in a meta-logic) hold. If the preconditions hold then the proof planner executes the post-conditions (which are also a sequence of statements in the meta-logic). This constructs the output goal or goals. A typical method is the `induction` method, whose input is a universally quantified goal, and whose preconditions then select a suitable induction variable, and induction scheme. The output of the `induction` method are appropriate base and step cases.

Proof planning offers several potential advantages over other theorem proving techniques for the task of transforming CSPs automatically. First, methods can be given very strong preconditions to limit the transformations to those that are likely to produce a problem that is simpler to solve. Second, methods can act at a very high level. For example, they can perform complex rewriting, simplifications, and transformations. Such steps might require very long and complex proofs to justify at the level of individual inference rules. And third, the search control in proof planning is cleanly separated from the inference steps. We can therefore try out a variety of search strategies like best-first search or limited discrepancy search.

3 Extensions to Proof Planning

Whilst proof planning has a number of features that make it well suited to the task of transforming CSPs, in constructing CGRASS we have had to extend it along a number of dimensions to deal with the following issues:

Non-monotonicity: Some of CGRASS’ methods add new constraints, whereas others replace a constraint by a tighter one, or eliminate redundant constraints. 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 less clear. We must decide when to stop making transformations and start searching 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 an executive in the style of a proof critic [6] which terminates CGRASS 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 CGRASS

The implementation (in Java) of CGRASS discussed here takes a problem instance as input. We are developing an implementation which will consider an entire problem class. CGRASS' input consists of a finite set of initial instance variables, each with an associated finite domain (either explicitly or as bounds) and constraints over these variables. Output is created in the same simple format. Hence, very little effort is necessary to translate CGRASS' output into the required input for a variety of existing solvers.

Internally, CGRASS works with a simplified syntax, not only to promote efficiency, but also to reduce the number and complexity of methods needed. For example, inequalities are always rearranged into the form $x < y$ or $x \leq y$. Hence the input to a method never has to match $x > y$ or $x \geq y$, halving the number of methods in some cases. A further example is that subtraction is replaced by a sum and a coefficient of -1. These restrictions are not placed upon the input.

4.1 Normalisation

CGRASS transforms the constraint set into a normal form, inspired by that used in the HARTMATH computer algebra system[1]. A normal form is necessary to deal with associative and commutative operators. It allows us, to an extent, to replace the test for semantic equivalence with a much simpler syntactic comparison. The normal form used is a combination of a lexicographic ordering within individual constraints and the constraint set as a whole, and certain simplifications.

We define a total order over the types of expressions that CGRASS supports. The constraint set is transformed into a minimal state with respect to this order. Constants are at the top of the order, followed by variables, fractions, sums and products. Further down the order are constraint types such as equalities, inequations, inequalities and

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

special constraints such as 'all-different'. Expressions of different type are ordered via their position in the order. Expressions of the same type are ordered recursively; each type has an associated method of self-comparison. The base case is where two constants or two variables are compared. In the former case, the comparison is by value, with least first and in the latter the comparison is lexicographically by name. Sums and products are represented in a 'flattened' form, hence their arguments are simply sorted using the above comparison to maintain a lexicographic order. Similarly, the lexicographically least side of an equation or inequation is forced to be the left hand side.

Consider the following example:

$$x_8 + x_7 \neq x_6 + x_5$$
$$x_4 * 2 + x_3 = 2 * x_1 + x_2$$

In normal form:

$$x_2 + 2 * x_1 = x_3 + 2 * x_4$$
$$x_5 + x_6 \neq x_7 + x_8$$

Equality is higher in the type order than an inequation, hence the re-ordering of the two constraints. The sums are ordered internally and recursively, then re-ordered as appropriate to the constraint.

Simplification procedures consist of the collection of like terms, cancellation and the removal of common factors. Consider the following example:

$$2 * 6 * x_1 + 4 * x_2 = 6 * x_1 + x_3 * 2 * 2 + 6 * x_1$$

Following lexicographical ordering, we collect the constants and occurrences of x_1:

$$4 * x_2 + 12 * x_1 = 4 * x_3 + 12 * x_1$$

Next we perform cancellation:

$$4 * x_2 = 4 * x_3$$

Finally, we remove the common factor:

$$x_2 = x_3$$

Lexicographic ordering and simplification are interleaved until no further change is possible. They reduce the workload of CGRASS substantially, both in providing a syntactic test for equality and avoiding such simplification routines being written as explicit methods. The latter saving is substantial: a larger method base means more work in matching against the constraint set at each iteration of the CGRASS inference loop. Normalisation, on the other hand, is performed immediately whenever possible.

5 Methods

We illustrate the methods currently implemented in CGRASS by means of a small instance of the Golomb ruler problem. Peter van Beek has proposed the Golomb ruler problem as a challenging constraint satisfaction problem for the CSPLib benchmark library (available as `prob006` at `http://www.csplib.org`). 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. The longest known optimal ruler has 21 marks and is of length 333. Such rulers have practical applications in radio astronomy. Smith et al. [8] used the Golomb ruler as the basis of an interesting exercise in modelling CSPs and identified a number of implied constraints by hand.

We begin with a concise model of the problem with n ticks represented by variables $x_1, ..., x_n$, each with domain $\{0, ..., n^2\}$:

$$\text{minimise: } max_i(x_i)$$
$$\{\forall i, j, k, l \in [1, n] : (x_i - x_j \neq x_k - x_l) \,|\, (i \neq j) \wedge (k \neq l) \wedge (i \neq k \vee j \neq l)\}$$

Taken literally, this is a poor model. The constraints are quaternary, and will be delayed by most solvers. There is also a large amount of symmetry present. However, it serves to illustrate how CGRASS can make a substantial improvement to a basic model.

We focus on the 3-tick ruler for the purpose of this example. The basic model produces 30 constraints. CGRASS' initial normalisation of the constraint set immediately reduces this number to 12. This is achieved in two ways. Firstly, constraints with reflection symmetry across an inequation are identical following normalisation, hence only one copy is kept. Secondly, some cancellation is possible, such as in the following case:

$$x_1 - x_2 \neq x_1 - x_3$$

Following cancellation and removal of the common factor, we get the much simpler form, to which several input constraints reduce.

$$x_2 \neq x_3$$

Hence, a large saving is made before CGRASS has performed any method application. Table 1 presents the state of the problem at this point. The 'minimise' statement is omitted throughout for brevity.

$x_1 \neq x_2$	$x_1 \neq x_3$	$x_2 \neq x_3$
$x_1 - x_2 \neq x_2 - x_1$	$x_1 - x_2 \neq x_2 - x_3$	$x_1 - x_2 \neq x_3 - x_1$
$x_1 - x_3 \neq x_2 - x_1$	$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$
$x_2 - x_1 \neq x_3 - x_2$	$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$

Table 1. 3-tick Golomb Ruler. Initial state following normalisation.

5.1 Symmetry

Often the most useful constraints can only be derived when some or all symmetry has been broken. Hence, CGRASS attempts to detect and break symmetry with new constraints as a pre-processing step. It begins by looking for symmetrical variables, i.e.

pairs of variables with identical domains such that, if all occurrences of this pair in the constraint set are exchanged and the constraint set is re-normalised, it returns to its original state. Candidate variables with a uniform number of occurrences in the constraint set are first grouped together before comparisons are made. The transitivity of symmetry is used to compare as small a number of pairs of variables as possible. Efficiency is further improved by making pairwise comparisons of normalised constraint sets.

This process partitions the variables into symmetry classes. The elements of each class are formed into a list, and ordered lexicographically. Symmetry is broken by adding weak inequalities between adjacent variables in each list, e.g. $x_1 \leq x_2, x_2 \leq x_3, ...$ We avoid adding constraints between all pairs, since simple bounds consistency maintains consistency on the transitive closure. The inequalities are not *implied* since they do not follow from the initial model. However, symmetry breaking constraints are useful both for reducing search and for generating further implied constraints.

Symmetry testing on our example reveals that the variables x_1, x_2 and x_3 are symmetrical, hence the constraints below are added:

$$x_1 \leq x_2$$
$$x_2 \leq x_3$$

It is also possible to identify symmetries among non-atomic terms. This is potentially an expensive process, hence CGRASS adopts a heuristic approach, only comparing terms that are likely to be symmetrical. These heuristics are based on *structural equivalence*. Two terms are structurally equivalent if they are identical when explicit variable names in each are replaced with a common indistinguishable marker. For example,

$$\frac{x_1}{x_2 * x_3}, \frac{x_4}{x_5 * x_6}$$

become:

$$\frac{\#}{\# * \#}, \frac{\#}{\# * \#}$$

and are therefore structurally equivalent. Each pair of variables, x_1 and x_4, x_2 and x_5, and x_3 and x_6 are exchanged throughout the constraint set before re-normalisation and a check for equivalence. This process does not reveal any further symmetries in the example problem, but is useful in general (see [4], for example).

CGRASS can now fire the `strengthenInequality` method, as presented in figure 1. This is one example of a number of simple but useful methods to which CGRASS ascribes a high priority during method selection. Other examples are various instances of the `nodeConsistency` and `boundsConsistency` methods which deal with the filtering of domain elements. 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.

Indeed, the `boundsConsistency` method can now fire, pruning the domains of x_1, x_2 and x_3 according to their strict ordering. This leaves the problem in the state as presented in table 2. Note that the constraint $x_1 \neq x_3$ is now redundant. One could foresee the addition of a relatively simple method that takes as input a set of strict inequalities and an inequation in order to detect and remove such a redundancy.

Preconditions:

1. There exist two expressions, $x \neq y$ and $x \leq y$

Postconditions:

1. Add a new constraint of the form, $x < y$
2. Delete $x \neq y$ and $x \leq y$

Fig. 1. The `strengthenInequality` method.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$x_1 - x_2 \neq x_2 - x_1$	$x_1 - x_2 \neq x_2 - x_3$	$x_1 - x_2 \neq x_3 - x_1$
$x_1 - x_3 \neq x_2 - x_1$	$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$
$x_2 - x_1 \neq x_3 - x_2$	$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$

Table 2. State following symmetry-breaking, bounds consistency.

5.2 Introduce

The model as it stands still contains 9 quaternary constraints. One powerful means of reducing the arity of these constraints is to introduce one or more new variables which the `eliminate` method (see below) then uses to replace sub-terms within them. Therefore, we have developed the `introduce` method, as presented in figure 2. Since this

Preconditions:

1. There exists a sub-term, *Exp*, with arity greater than 1 that occurs more than once.
2. *someVariable* = *Exp* is not already present.

Postconditions:

1. Generate a new variable, x, with domain derived from *Exp*.
2. Add a constraint of the form $x =$ *Exp*.

Fig. 2. The `introduce` method.

method introduces new terms, it has a potentially explosive effect. CGRASS therefore only attempts to apply it when all the simpler methods, which tend to have a reductive effect, are inapplicable. In addition, complex preconditions are attached to `introduce` to prevent its application unless there is strong evidence that the new variable will be useful. First of all, we insist that the sub-term, *Exp*, under consideration has an arity (i.e. the number of different variables it contains) of at least 2: there is little point in adding a new variable which is a simple expression of one already in existence.

Secondly, variables with a higher number of occurrences have a wider reaching effect when propagation is performed on them. We require *Exp* to occur at least twice in the constraint set before it can be considered for replacement by a variable. Finally, we check that some other variable is not already defined to be equal to *Exp*. If these conditions are met, CGRASS generates a new variable, x, calculating the bounds of its domain from the upper and lower bounds on *Exp*.

5.3 Eliminate

The sub-term $x_1 - x_2$ in the example meets the input preconditions of `introduce`. CGRASS binds a new variable, z_0, to it with domain $\{$-8 .. 6$\}$. In order to make use of z_0, however, the companion `eliminate` method is necessary. There are multiple versions of `eliminate`, using equalities (figure 3) and inequalities to perform Gaussian-like elimination of a particular sub-term.

Preconditions:

1. There exists a constraint *Lhs* = *CommonExp* such that *CommonExp* is also present in a constraint, c, in the constraint set.
2. Constraint c_{new} is obtained by replacing all occurrences of *CommonExp* by *Lhs* in c.
3. The complexity of c_{new} is less than that of c.
4. c_{new} is not obviously redundant.

Postconditions:

1. Add c_{new} to the constraint set.
2. Remove c from the constraint set.

Fig. 3. The `Eliminate(equality)` method.

As per `introduce`, the uncontrolled application of `eliminate` can result in an explosion in the size of the constraint set, hence the strong pre- and post-conditions on this method. The resulting constraint must be of lower complexity (i.e. smaller number of constituent terms) than the original. Also, we perform simple checks for redundancy such as 0 arity (e.g. $1 < 2$) or, in the case of equality, the left hand side being equal to the right hand side. Finally, when eliminating with equality the original constraint is removed following elimination in order to avoid cluttering the constraint set. `Eliminate` is one of the methods that must be exhausted before `introduce` can fire again. Since `eliminate` simplifies the constraint set, it reduces the chance of sub-terms recurring, preventing the preconditions of `introduce` being met prematurely.

Following the introduction of $z_0 = x_1 - x_2$ in the example, various instances of `eliminate` can fire. For instance, we can eliminate x_1 in favour of x_2 in this equation using $x_1 < x_2$ to give: $z_0 < 0$. This is a unary constraint which is immediately used to trigger the `nodeConsistency` method, reducing the domain of z_0 to {-8 .. 0}. We can also substitute z_0 into a number of the quaternary inequations, reducing the complexity of each. This leaves the problem in the state presented in table 3.

$x_1 \in \{0..7\}$	$x_2 \in \{1..8\}$	$x_3 \in \{2..9\}$
$x_1 < x_2$	$x_2 < x_3$	$x_1 \neq x_3$
$z_0 = x_1 - x_2$	$z_0 \neq x_2 - x_3$	$z_0 \neq x_3 - x_1$
$x_1 - x_3 \neq x_3 - x_1$	$x_1 - x_3 \neq x_3 - x_2$	$x_1 - x_3 \neq -z_0$
$x_2 - x_3 \neq x_3 - x_1$	$x_2 - x_3 \neq x_3 - x_2$	$x_3 - x_2 \neq -z_0$

Table 3. State following introduction of and elimination with z_0.

5.4 All-different

CGRASS now introduces, and eliminates with, a further two variables, $z_1 = x_2 - x_3$ and $z_2 = x_3 - x1$. This leads to the much-improved situation presented in table 4. One

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

Table 4. State following introduction of and elimination with z_0.

further method available to CGRASS is `genAllDiff` which, as its name suggest, attempts to generate an all-different constraint from a clique of not-equals constraints. An all-different constraint is desirable because of the powerful propagation methods available for it within constraint solvers [7]. Since maximal-clique identification is an NP-complete problem, CGRASS uses a greedy procedure to quickly find as large a clique as possible. Typically this is the maximal clique.

The `genAllDiff` method takes as input the subset of all inequations with single variables on both the left and right hand sides. The greedy procedure traverses this subset, generating a list for each variable of the variables with which it is not equal. Starting with those variables associated with the largest lists, it attempts to determine whether they form a clique. If a variable is not part of the current clique it is thrown out. If the clique is reduced to a single variable, the procedure looks for the variables associated with the next largest-sized lists, and so on.

In the example, `genAllDiff` successfully replaces the inequations involving the z variables with a single all-different constraint. This leads to the final problem state (see table 5). This method has a lower priority than `introduce`: waiting for `introduce` to be exhausted maximises the chance of finding the largest clique of inequations.

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

Table 5. Final state.

6 Results

We compared the performance of Ilog Solver on basic and transformed models of 5 instances of the Golomb ruler problem. The transformed models are similar to the one presented in table 5. The results are presented in table 6. The smallest instances are so easy to solve that it is not worth the effort of transformation. For larger instances, however, the transformed model becomes significantly easier to solve, with the gap in performance increasing rapidly with n.

The size of the input generated from the basic model also increases dramatically with n. For instance, $n = 6$ generates 870 constraints. This is accompanied by a marked increase in the time required by CGRASS to make the transformations. On the small instances tested, this means that the total time for transformation and solution exceeds the time for solution of the basic model alone. However, the time required by CGRASS as n increases grows more slowly than the time taken to solve the basic model. Hence, at larger (and therefore more interesting) values of n, a net benefit can be expected.

Model		3 ticks	4ticks	5 ticks	6 ticks	7 ticks
Basic	Fails	10	103	3632	111094	-
	Choice-points	12	107	3637	111101	
	Time	0.06s	0.06s	0.4s	28.8s	
Transformed	Fails	3	7	55	374	3611
	Choice-points	5	10	59	381	3618
	Time	0.05s	0.05s	0.06s	0.1s	0.3s

Table 6. Results: Golomb Ruler. Dash indicates problem unsolved within 1 hour.

One means of overcoming the problem of overwhelming CGRASS with a large number of constraints derived from a basic model is to use it interactively. Given the final model of the 3-tick ruler, it is not difficult for a human to see how this model could be generalised to n ticks. The comparative results of the basic and final models presented in table 6 indicate that the effort expended on such a process could easily be justified as n grows larger.

7 Reasoning at Higher Levels of Abstraction

If we wish to make automatic transformations efficiently, however, we must lift our reasoning to a higher level. Direct support for quantified constraint expressions would immediately reduce the size of the input to two constraints in our example:

$$\text{minimise: } max_i(x_i)$$
$$\{\forall i, j, k, l \in [1, n] : \ (x_i - x_j \neq x_k - x_l) \,|\, (i \neq j) \wedge (k \neq l) \wedge (i \neq k \vee j \neq l)\}$$

Although writing methods would typically be complicated by the need to support quantified constraints, some transformations are very straightforward at this level of abstraction. For example, the `genAllDiff` method requires just the following simple input.

$$\forall i, j \ i < j \rightarrow x_i \neq x_j$$

Compare this with the relatively complicated process outlined in section 5.4.

A further benefit is the ability to reason about an entire class of problems rather than particular instances. The class of Golomb Ruler problems, parameterised by the number of ticks n, can be described by the pair of constraints above. Hence, all transformations made at this level of abstraction are valid for the whole problem class. This is clearly more efficient than repeating a large amount of work for each instance under consideration. This is not to say that we should abandon reasoning about problem instances altogether, however. It is likely that some transformations will be valid only for a subset of the instances of a class. Therefore, we intend to extend (rather than replace) the range of transformations CGRASS can make to support transformations at the level of abstraction at which they are most straightforward.

7.1 A Higher Level Input Language

Even following the introduction of quantified constraint expressions, some transformations remain difficult. One of the key transformations made in section 5 was to recognise the symmetry of the tick variables and break this symmetry via the introduction of weak inequalities. By inspection, it is clear that detecting this symmetry given only the quantified problem representation is a difficult task. For this reason, we believe that a higher level input language, perhaps along the lines of OPL [10], is necessary.

This section describes how we might transform a high level description of the Golomb Ruler problem into (a family of) instances suitable for input to a constraint solver. We begin with an English description of this problem:

- Put n ticks on a ruler of size m such that all the inter-tick distances are unique. Minimise m.

We cannot expect CGRASS to work with this level of input, hence the user must make the initial transformation shown below. We argue that this is less work than producing the quantified input statement used previously. It is certainly more natural to be able to write a model in terms of 'distance'.

1. Find $T \subseteq \{0, ..., m\}$ subject to:
2. Minimise m
3. $|T| = n$
4. $\{$distance$(\{x, y\}) \neq$ distance$(\{x', y'\})| \{x, y\} \subseteq T, \{x', y'\} \subseteq T, \{x, y\} \neq \{x', y'\}\}$
5. distance$(\{x, y\}) = |x - y|$

Note that $\{x, y\}$ denotes a set of size 2.

This description is not a constraint satisfaction problem, hence it is not yet suitable for input to a system like Ilog's Solver. We will use *refinement* rules in order to move to lower levels of abstraction as necessary. A straightforward refinement from this initial representation would produce a reasonable input model. Already, we gain over previous attempts because indices into a set are not defined. This is a symmetry we can break via inequalities as we create it, when creating a variable per element. However, we wish to continue to work at this high level for as long as possible so that our transformations hold for all Golomb rulers.

Given the recurrence of distance$(\{x, y\})$, it is natural to introduce a set of distance variables.

6. $\{d_{xy} =$ distance$(\{x, y\}) \mid \{x, y\} \subseteq T\}$

Substituting (6) into (4) gives:

7. $\{d_{xy} \neq d_{x'y'} \mid \{x, y\} \subseteq T, \{x', y'\} \subseteq T, \{x, y\} \neq \{x', y'\}\}$

It should not be difficult to notice that (7) defines a clique. A lifted version of our all-different introduction method would produce:

8. all-diff$(\{d_{xy} | \{x, y\} \subseteq T\})$

Next, we need a general refinement rule:

– To pick a subset of size n from a set A of size m, totally ordered by $\leq$, introduce a set S of n variables, $\{s_1, ..., s_n\}$. The domain of each s_i is A. We could conceivably build in the simple bounds consistency argument into this, e.g. $s_1 \leq m - n + 1$. Assignments to variables in S must also form a set totally ordered by $\leq$ with respect to their subscripts. That is, we have constraints $s_1 < s_2 < ... < s_n$.

Applying this rule:

9. Assign S=$\{s_1, s_2, ..., s_n\}$, where each s_i has domain $\{0, ..., m\}$.
   $s_1 < s_2 < ... < s_n$.
10. $\{d_{ij} =$ distance$(\{s_i, s_j\}) \mid \{s_i, s_j\} \subseteq S\}$
11. all-diff$(\{d_{ij} | \{s_i, s_j\} \subseteq S\})$

The domain of each s_i is bounded above by m which, from (2), we are trying to minimise.

12. Minimise(maximal element of S)

Straightforwardly from (9) and (12):

13. Minimise(s_n)

Substituting (5) into (10) gives:

14. $\{d_{ij} = |s_i - s_j| \mid \{s_i, s_j\} \subseteq S\}$

Refining (11) and (14) from sets to ordered pairs:

15. all-diff($\{d_{xy}| < x, y > \subseteq S \times S, i < j\}$)
16. $\{d_{ij} = |s_i - s_j| \mid < s_i, s_j > \subseteq S \times S, i < j\}$

In (16) we have $s_i < s_j$. From the semantics of absolute, we can re-write:

14. $\{d_{ij} = s_j - s_i \mid < s_i, s_j > \subseteq S \times S, i < j\}$

The final problem state is as follows. It is equivalent to the final representation obtained in section 5, but holds for the entire problem class. Symmetry amongst the tick variables has been broken, and all-different difference variables for the inter-tick distances have been introduced.

- Assign S=$\{s_1, s_2, ..., s_n\}$, where each s_i has domain {0, ..., m}.
  $s_1 < s_2 < ... < s_n$.
- Minimise(s_n)
- $\{d_{ij} = s_j - s_i \mid < s_i, s_j > \subseteq S \times S, i < j\}$
- all-diff($\{d_{xy}| < x, y > \subseteq S \times S, i < j\}$)

Working with a higher level input language has allowed us to avoid the problem of detecting symmetry by making use of sets. We have also retained the advantage of making valid transformations for the whole problem class. Furthermore we avoided the overwhelming size of the input that inevitably causes problems when transforming naive representations of individual instances. The use of refinement rules allows us to move to progressively more concrete levels of abstraction as necessary to perform transformations at the appropriate level.

8 Conclusions

We have described CGRASS, a system for the automatic transformation of a naive model of a constraint satisfaction problem into one that requires significantly less effort to solve. CGRASS adopts a proof planning style architecture, transforming a model via the application of methods which encapsulate modelling expertise. The open ended nature of the search for a good model necessitated several extensions to standard proof planning, such as the ability to support non-monotonicity and prevent looping. The set of methods described here should be viewed as a representative sample. It is not complete in any sense, and we will continue to extend it in future.

The current implementation of CGRASS is able to transform individual problem instances only. Results obtained from experiments on the Golomb Ruler problem suggest that this approach is impractical in general. We have discussed the need for the ability to reason at multiple levels of abstraction, since attempting a transformation at the 'wrong' level can prove extremely difficult. We have outlined how the Golomb Ruler might be effectively transformed from a high level description into a good model. A principle element of future work is to extend CGRASS' set of methods to allow it to reason at higher levels of abstraction. In addition, we will introduce refinement rules to move from higher to lower levels of abstraction as necessary.

As a further piece of future work, we will also consider whether CGRASS' methods can be implemented via constraint handling rules [5]. Possible obstacles include more complex pattern matching, the use of best-first search as the method base grows in complexity, and the use of an executive which decides when to stop inferring and start searching.

Acknowledgements This project is supported by EPSRC Grant GR/N16129[2]. The authors wish to thank Brahim Hnich for discussions about generating implied constraints automatically. Julian Richardson's adaptation of PRESS to manipulate inequalities and its success at generating a number of implied constraints automatically influenced our approach.

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. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M.E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. LNAI 449.
3. M.D. Ernst, T.D. Millstein, and D.S. Weld. Automatic SAT-compilation of planning problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, pages 1169–1176, 1997.
4. A.M. Frisch, I. Miguel, and T. Walsh. Extensions to proof planning for generating implied constraints. In *Proceedings of Calculemus-01*, pages 130–141, 2001.
5. T. Frühwirth. Theory and practice of constraint handling rules. In P. Stuckey and K. Marriot, editors, *Journal of Logic Programming, Special Issue on Constraint Logic Programming*, volume 37(1–3), pages 95–138, 1998.
6. A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proof. In *Proceedings of LPAR'92, Lecture Notes in Artificial Intelligence 624*. Springer-Verlag, 1992. Also available as Research Report 592, Dept of AI, Edinburgh University.
7. J.C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of the 12th National Conference on AI*, pages 362–367. American Association for Artificial Intelligence, 1994.
8. B. Smith, K. Stergiou, and T. Walsh. Using auxiliary variables and implied constraints to model non-binary problems. In *Proceedings of the 16th National Conference on AI*, pages 182–187. AAAI, 2000.
9. 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.
10. P. van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

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