Conditional Symmetry Breaking [⋆]

Ian P. Gent[1], Tom Kelsey[1], Steve A. Linton[1], Iain McDonald[1], Ian Miguel[1], Barbara M. Smith[2]

[1]School of Computer Science, University of St Andrews, St Andrews, Fife, UK
[2]Cork Constraint Computation Centre, University College Cork, Cork, Ireland
{ipg,tom,sal,iain,ianm}@dcs.st-and.ac.uk, bms@4c.ucc.ie

Abstract. We introduce the study of *Conditional* symmetry breaking in constraint programming. This arises in a sub-problem of a constraint satisfaction problem, where the sub-problem satisfies some *condition* under which additional symetries hold. Conditional symmetry can cause redundancy in a systematic search for solutions. Breaking this symmetry is an important part of solving a constraint satisfaction problem effectively. We demonstrate experimentally that three methods, well-known for breaking unconditional symmetries, can be applied to conditional symmetries. These are: adding conditional symmetry-breaking constraints, reformulating the problem to remove the symmetry, and augmenting the search process to break the conditional symmetry dynamically through the use of a variant of Symmetry Breaking by Dominance Detection (SBDD).

1 Introduction

Constraint programming has been used successfully to tackle a wide variety of combinatorial problems. To apply constraint programming to a particular domain, the problem must be *modelled* as a constraint program. Typically, many alternative models exist for a given problem, some of which are more effective than others. Constructing an effective constraint model is a difficult task.

An important aspect of modelling is dealing with *symmetry.* Symmetry in a model can result in a great deal of wasted effort when the model is solved via systematic search. To avoid this, the symmetry must be broken effectively. Most research on symmetry in constraint models considers only the symmetry present in a model before search begins. As we will discuss, symmetries can often form during search. We call this *conditional* symmetry, since its formation depends on the choices made during search. To avoid redundant search, it is important to break this symmetry also.

This paper discusses three ways to deal with conditional symmetry. First, we can add constraints to a model to detect and break the symmetry as it arises.

[⋆] We thank Alan Frisch and Chris Jefferson. Ian Gent is supported by a Royal Society of Edinburgh SEELLD/RSE Support Research Fellowship. Ian Miguel is supported by a UK Royal Academy of Engineering/EPSRC Research Fellowship. This material is based in part on works supported by the Science Foundation Ireland under Grant No. 00/PI.1/C075.

Second, we can reformulate our model so that the new model does not have the conditional symmetry. Finally, we discuss how conditional symmetry can be broken during search.

2 Background

The finite domain *constraint satisfaction problem* (CSP) consists of a triple $\langle X, D, C\rangle$, where X is a set of variables, D is a set of domains, and C is a set of constraints. Each $x_i \in X$ is associated with a finite domain $D_i \in D$ of potential values. A variable is *assigned* a value from its domain. A constraint $c \in C$, constraining variables $x_i, \ldots, x_j$, specifies a subset of the Cartesian product $D_i \times \ldots \times D_j$ indicating mutually compatible variable assignments. A *constrained optimisation problem* is a CSP with some objective function, which is to be optimised.

A *partial assignment* is an assignment to one or more elements of X. A solution is a partial assignment that includes all elements of X. This paper focuses on the use of systematic search through the space of partial assignments to find such solutions. A sub-CSP, P', of a CSP P is obtained from P by adding one or more constraints to P. Note that assigning a value v to a variable x is equivalent to adding the constraint $x = v$.

A *symmetry* in a CSP is a bijection mapping solutions to solutions and non-solutions to non-solutions. A *conditional symmetry* of a CSP P holds only in a sub-problem P' of P. The conditions of the symmetry are the constraints necessary to generate P' from P. Conditional symmetry is a generalisation of unconditional symmetry, since unconditional symmetry can be seen as a conditional symmetry with an empty set of conditions. We focus herein on conditions in the form of partial assignments.

3 Conditional Symmetry-breaking Constraints

A straightforward method of breaking conditional symmetries is to add constraints to the model of the form: *condition* $\rightarrow$ *symmetry-breaking constraint* where *condition* is a conjunction of constraints, for instance a partial assignment such as $x = 1 \land y = 2$, that must be satisfied for the symmetry to form. As in unconditional symmetry breaking [2], the symmetry-breaking constraint itself usually takes the form of an ordering constraint on the conditionally symmetric objects. We report case studies of breaking conditional symmetry in this way.

3.1 Graceful Graphs

The first case study is of conditional symmetry in finding all graceful labellings [6] in a class of graphs. A labelling f of the vertices of a graph with e edges is *graceful* if f assigns each vertex a unique label from $\{0, 1, ..., e\}$ and when each edge xy is labelled with $|f(x) - f(y)|$, the edge labels are all different.

(Hence, the edge labels are a permutation of 1, 2, ..., e.) Finding a graceful labelling of a given graph, or proving that one does not exist, can easily be expressed as a constraint satisfaction problem. The CSP has a variable for each vertex, $x_1, x_2, ..., x_n$ each with domain $\{0, 1, ..., e\}$ and a variable for each edge, $d_1, d_2, ..., d_e$, each with domain $\{1, 2, ..., e\}$. The constraints of the problem are that: if edge k joins vertices i and j then $d_k = |x_i - x_j|$; $x_1, x_2, ..., x_n$ are all different; and $d_1, d_2, ..., d_e$ are all different (and form a permutation).

The graph shown in Figure 1 is an instance of a class of graphs listed in Gallian's survey [6] as $C_n^{(t)}$: they consist of t copies of a cycle with n nodes, with a common vertex. For $n = 3$, these graphs are graceful when $t \equiv 0$ or 1 (mod 4).The nodes are numbered to show the numbering of the variables in the CSP model, i.e. node 0 is the centre node, represented by the variable x_0.

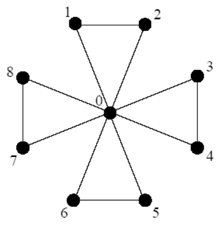

Fig. 1. The windmill graph $C_3^{(4)}$

The symmetries of the CSP are (i) swapping the labels of the nodes other than the centre node in any triangle, e.g. swapping the labels of nodes 1 and 2; (ii) permuting the triangles, e.g. swapping the labels of nodes 1 and 2 with those of nodes 3 and 4; (iii) changing every node label x_i for its complement $e - x_i$.

It is easy to show that the centre node cannot have a label > 1 and $< e - 1$, where e is the number of edges. Since there must be an edge connnecting two nodes labelled 0 and e, if the centre node's label is not 0 or e, then two other nodes in a triangle, e.g. nodes 1 and 2, must be labelled 0 and e. But then, unless the centre node is labelled 1 or $e-1$ there is no way to get an edge labelled $e-1$, given that the largest node label is e. The labels 0, 1, $e - 1$ and e are possible for the centre node, however, if there is a graceful labelling.

Suppose we have a graceful labelling of a graph in this class, with the centre node labelled 0. In any triangle, where the other two nodes are labelled a and b, with $a < b$, we can replace a with $b - a$ to get another solution. The edge labels in the triangle are permuted as shown in Figure 2. Any graceful labelling of $C_3^{(t)}$ with centre node labelled 0 has 2^t equivalent labellings by changing or not changing the labels within each of the t triangles in this way. The effect of an instance of this conditional symmetry, on nodes 0, 1, 2, say, depends on whether node 0 is labelled 0 and which of nodes 1 and 2 has the smaller label; hence, we need to know the assignments to these three variables. A graceful labelling

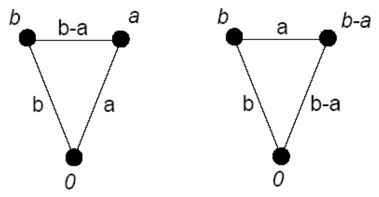

Fig. 2. Relabelling a triangle in a graceful labelling with centre node labelled 0

with the centre node labelled 1 can be transformed into an equivalent labelling similarly: a triangle labelled 1, a, b, with $a < b$ can be relabelled 1, $b - a + 1$, b. Again, this is conditional on the three assignments. There are equivalents for the other possible labels for the centre node, $e - 1$ and e.

In a labelling with the centre node labelled 1, there must be a triangle labelled 1, 0, e, since there has to be an edge whose endpoints are labelled 0 and e. The remaining nodes have labels in the range 3, .., $e - 1$. (Since we already have an edge labelled 1, we cannot have a node labelled 2, since it has to be connected to the centre node.)

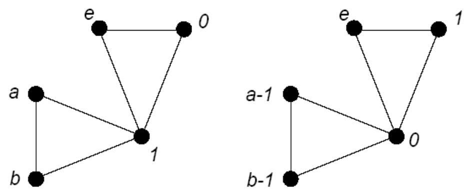

Fig. 3. Transforming a labelling with centre node labelled 1.

Figure 3 (left) shows the 1, 0, e triangle and another representative triangle. We can transform the labels of all the nodes as shown in the right-hand figure. If the original labelling is graceful, so is the transformed labelling. Hence, any labelling with centre node labelled 1 is equivalent to one with centre node labelled 0. Note that the reverse is not true: if the centre node is 0, there need not be a triangle labelled 0, 1, e.

Hence, there are two conditional symmetries. One has precondition $x_0 = 1$ and its effect is to transform the node labels $0, 1, 2, 3, ..., e{-}1, e$ into $1, 0, 2, 2, ..., e-2, e$ respectively. (The effect on a node label of 2 is irrelevant, because it cannot occur.) The other has preconditions $x_0 = 0$ *and* $x_1 = 1, x_2 = e$ *or* $x_3 = 1, x_4 = e$ *or* ... The effect of the symmetry is again easily expressed as a permutation of the values: $0, 1, 2, 3, ...e-2, e-1, e$ become $1, 0, 3, 4, ...e-1, e-1, e$. Similarly, if the centre node is labelled $e{-}1$, we can transform any resulting graceful labelling into one with the centre node labelled e.

Ignoring the conditional symmetries for now, the symmetries of the CSP can easily be eliminated by adding constraints to the model.

– In each triangle, we can switch the labels of the nodes that are not the central node. Constraints to eliminate this are: $x_{2i-1} < x_{2i}, i = 1, 2, ..., t$
– We can permute the triangles. Given the previous constraints, we can add the following to eliminate this: $x_{2i-1} < x_{2i+1}, i = 1, 2, ..., t-1$
– To eliminate the complement symmetry, we can post: $x_0 < e/2$.

The conditional symmetries can also be eliminated easily. First, the conditional symmetry in the labellings where the central node is 0 requires knowing which of the two other nodes in each triangle has the smaller label. Because of the constraints just added, it is the first one. In terms of Figure 3, we choose the labelling 0, a, b for the triangle and want this to be lexicographically smaller than 0, $b-a$, b. We can add a conditional constraint: if $x_0 = 0$, then $2x_{2i-1} < x_{2i}$, for $i = 1, 2, ..., t$. We have shown that the labellings with the central node labelled 1 are equivalent to some of the labellings with node 0 labelled 0. Further, 0 and 1 are the only possible labels for the central node, given the constraint to eliminate the complement symmetry. Hence, we can simply add $x_0 = 0$ to eliminate this conditional symmetry. This simplifies the conditional constraints given earlier: since we know that $x_0 = 0$, we can drop the condition and just have $2x_{2i-1} < x_{2i}$, for $i = 1, 2, ..., t$. Hence, in this example, all the symmetries, including the conditional symmetries, can be eliminated by simple constraints.

Using the symmetry-breaking constraints to eliminate just the graph and complement symmetries, the graph in Figure 1 has 144 graceful labellings. Eliminating the conditional symmetries reduces these to 8. The resulting reduction in search would be greater still for larger graphs in the same class, $C_3^{(t)}$. This case study demonstrates that eliminating conditional symmetry can sometimes be done with little overhead and reduce the search effort enormously.

3.2 The Patience/Solitaire game 'Black Hole'

We now show the value of conditional symmetries in a case study of the game 'Black Hole'. Different approaches to solving this game are described in [7]. It was invented by David Parlett with these rules:

"*Layout* Put the Ace of spades in the middle of the board as the base or 'black hole'. Deal all the other cards face up in seventeen fans [i.e. piles] of three, orbiting the black hole.

"*Object* To build the whole pack into a single suite based on the black hole.

"*Play* The exposed card of each fan is available for building. Build in ascending or descending sequence regardless of suit, going up or down ad lib and changing direction as often as necessary. Ranking is continuous between Ace and King." [11]

The table below shows an instance of the game: the 18 columns represent the $A\spadesuit$ in the black hole and the 17 piles of 3 cards each.

	4♢	7♡	7♠	3♢	5♠	T♣	6♠	J♣	J♠	9♢	7♢	2♣	3♡	7♣	3♠	6♢	9♣
	9♠	9♡	J♡	4♠	K♢	Q♢	T♠	T♢	A♣	Q♠	K♠	Q♡	5♡	K♣	8♡	J♢	2♢
A♠	8♠	5♢	2♡	5♣	T♡	3♣	8♣	A♡	2♠	K♡	Q♣	4♡	6♣	6♡	A♢	4♣	8♢

and a solution to this game is:

A♠-2♣-3♠-4♢-5♠-6♠-7♠-8♡-9♠-8♠-9♣-T♠-J♠-Q♡-J♡-T♣-J♣-Q♢-
-K♢-A♣-2♠-3♡-2♢-3♣-4♡-5♡-6♣-7♡-8♣-7♣-6♢-7♢-8♢-9♡-T♡
-9♢-T♢-J♢-Q♠-K♠-A♡-K♡-Q♣-K♣-A♢-2♡-3♢-4♠-5♣-6♡-5♢-4♣

We can see conditional symmetry in Black Hole from the example. The first two piles both have 9s in the middle. If, at some point in the game, both 4♢ and 7♡ have been played, the two 9s are interchangeable *provided that* we don't need to play 9♠ before 9♡ to allow access to 8♠, or 9♡ before 9♠ to access 5♢. That is, the 9s are interchangeable if they are both played after both of their predecessors and before either of their successors. In these circumstances, we can choose the order of the two 9s and not backtrack on this choice.

We can represent a solution to the game as a sequence of the 52 cards in the pack, starting with A♠, the sequence representing the order in which the cards will be played into the Black Hole. The game can be modelled as a permutation problem: if the cards are numbered 0 (the A♠) to 51, the sequence of cards can be represented as a permutation of these numbers. There are two sets of dual variables: x_i represents the ith position in the sequence, and its value represents a card; y_j represents a card and its value is the position in the sequence where that card occurs. We have the usual channelling constraints: $x_i = j$ iff $y_j = i$, $0 \leq i, j \leq 51$. We set $x_0 = 0$.

The constraints that a card cannot be played before a card above it has been played are represented by $<$ constraints on the y_j variables. The constraints that each card must be followed by a card whose value is one higher or one lower are represented by constraints between x_i and x_{i+1} for $0 \leq i < 51$.

The variables $x_0, x_1, ..., x_{51}$ are the search variables: the variables $y_0, y_1, ..., y_{51}$ get assigned by the channelling constraints. The x_i variables are assigned in lexicographic order, i.e. the sequence of cards is built up consecutively from start to finish. The value ordering chooses cards. The top or middle layers are chosen before cards of the same rank lower down in the initial piles, and ties are broken by choosing cards in increasing rank order and an arbitrary suit order (♠, ♡, ♢, ♣). This fits with the problem, in that it makes sense to clear off the top layer of cards as quickly as possible. This simple model using only binary constraints models the problem successfully, but in practice search is prohibitive. We need other techniques to make search practical.

We now deal with conditional symmetry. Recall that in the example 9♠ and 9♡ are interchangeable if both have been played after the cards above them, 4♢ and 7♡, and before the cards immediately below them, 8♠ and 5♢. To break this conditional symmetry, we can add the constraint: if 4♢ $<$ 9♡ and 9♠ $<$ 5♢ then 9♠ $<$ 9♡. This constraint forces 9♠ to be played before 9♡ when they are interchangeable. Based on the initial layout, all constraints of this form can be added, pairwise, before search. The constraints are simplified if the preferred card of the pair is at the top of its pile or the other card is at the bottom of its pile, or both. The conditional symmetry-breaking constraints are designed to respect the value ordering; the same order of cards of each rank is preferred by both. Hence, the solution found is the same as the solution that would be found without the constraints. The constraints simply prevent the search from exploring subtrees that contain no solution. Hence, the number of backtracks with the constraints is guaranteed to be no more than without them. Furthermore, they appear to add little overhead in terms of runtime; they cannot become active until their condition becomes false on backtracking, and they then become simple precedence constraints that are cheap to propagate.

Our CP model was implemented in ILOG Solver 6. We used a benchmark set of 2,500 games, of which 2,189 are winnable [7]. With the conditional symmetry-breaking constraints, the CP model was highly effective at solving these instances. The longest run-time was 1,454sec. (on a 1.7GHz Pentium M PC, running Windows 2000). The distribution was very skewed; 97.5% of instances were solved in 20.8 sec. or less. All the instances were solved in a total of less than 11,000 sec. We also solved the first 150 instances of the 2,500 without conditional symmetry breaking, using a cut-off of 1,500 sec. (i.e. more than long enough to solve any of the instances with symmetry breaking). 20 instances timed out, and the total time to solve the 150 instances was over 37,000 sec. Some instances could still be solved with very little search; even so, the median backtracks increased from 81.5 to 3,618. Overall, it is not practicable to use the CP model to solve random instances of Black Hole without conditional symmetry breaking.

3.3 Steel Mill Slab Design

Our next case study is the steel mill slab design problem [5] (problem 38 at www.csplib.org). Steel is produced by casting molten iron into slabs. A finite number, σ, of *slab sizes* is available. An order has two properties, a *colour* corresponding to the route required through the steel mill and a *weight*. The problem is to pack the d input orders onto slabs so that the total slab capacity is minimised. There are two types of constraint: *Capacity constraints* specify that the total weight of orders assigned to a slab cannot exceed the slab capacity. *Colour constraints* specify that each slab can contain at most p of k total colours (p is usually 2). These constraints arise because it is expensive to cut the slabs up to send them to different parts of the mill.

We use a matrix model to represent this problem. Assuming the largest order is smaller than the largest slab, at most d slabs are required. Hence, a one-dimensional matrix of size d, $slab_M$, can be used to represent the size of each slab, a size of zero indicating that this particular slab is unused. A $d \times d$ 0-1 matrix, $order_M$, is used to represent the assignment of orders to slabs; $order_M[i, j] = 1$ if the ith order is assigned to the jth slab. Constraints on the rows ensure that the slab capacity is not exceeded:

$$\forall j \in \{1..d\} : \sum_{i \in \{1..d\}} weight(i) \times order_M[i, j] \leq slab_M[j]$$

where *weight*(i) is a function mapping the ith order to its weight. Constraints on the columns ensure that each order is assigned to one and only one slab:

$$\forall i \in 1..d : \sum_{j \in \{1..d\}} order_M[i,j] = 1$$

A second 0-1 matrix, $colour_M$ with dimensions $k \times d$, relates slabs and colours. A '1' entry in the ith column and jth row indicates that the ith colour is present on the jth slab. Constraints link $order_M$ and $colour_M$:

$$\forall i \in \{1..d\} \forall j \in \{1..d\} : order_M[i,j] = 1 \rightarrow colour_M[colour(i), j] = 1$$

where *colour*(i) is a function mapping the ith order to its colour. Constraints on the rows of $colour_M$ ensure that each slab is given no more than p colours.

$$\forall j \in \{1..d\} : \sum_{i \in \{1..k\}} colour_M[i,j] \leq p$$

In this initial formulation, there is a symmetry involving $slab_M$ and the rows of $order_M$: a solution can be transformed into a solution by permuting the values assigned to each element of $slab_M$ and permuting the corresponding rows of $order_M$. This symmetry can be broken by forming d *slabAndOrderRow* vectors, where the first element of *slabAndOrderRow*$[i]$ is $slab_M[i]$ and the remaining elements are the ith row of $order_M$, and lexicographically ordering as follows:

$$slabAndOrderRow[1] \geq_{\text{lex}} slabAndOrderRow[2] \geq_{\text{lex}} \ldots slabAndOrderRow[d]$$

Furthermore, $order_M$ has partial column symmetry. If two orders have equal weight and colour, the associated columns can be exchanged. This symmetry can be broken by combining symmetric orders into a single column, whose sum is constrained to be equal to the number of orders it represents.

There is a further symmetry conditional on the way that orders are assigned to slabs. Consider 3 'red' orders, order a of weight 6 and two instances of order b, with weight 3 (the last two are represented by a single column), and the following partial assignments to $order_M$:

$$\begin{array}{c|ccc|} & a & b & \ldots \\ slab_1 & 1 & 0 & \ldots \\ slab_2 & 0 & 2 & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{array} \qquad \begin{array}{c|ccc|} & a & b & \ldots \\ slab_1 & 0 & 2 & \ldots \\ slab_2 & 1 & 0 & \ldots \\ \ldots & \ldots & \ldots & \ldots \end{array}$$

These assignments are symmetrical. Note that the symmetry is conditional on both instances of b being assigned to the same slab, effectively creating a single 'super' order symmetrical to a. This is the simplest case of *compound* order symmetry, where individual orders combine to become symmetrical to single larger orders or other compounds.

To break compound order symmetry, we must know when and where the symmetry forms. For simplicity, we discuss only compound orders composed from multiple instances of the same order; the encoding can be extended straightforwardly to compounds formed from orders of different sizes. Consider an instance with 6 red orders of size 1. The assignment of these orders to slabs is represented by a single column of $order_M$, whose sum is constrained to be six. Up to two red compound orders of size three can form from the six red orders. Figure 4 (Top) presents example cases for which we must cater. In each example all the orders have been assigned to a slab, but in some cases one (Fig.4*d*, Fig.4*e*, Fig.4*f*) or both (Fig.4*a*) compounds have not formed.

$$a)\begin{pmatrix}1\\1\\1\\1\\1\\1\end{pmatrix}\; b)\begin{pmatrix}3\\3\\0\\0\\0\\0\end{pmatrix}\; c)\begin{pmatrix}0\\0\\6\\0\\0\\0\end{pmatrix}\; d)\begin{pmatrix}1\\1\\1\\3\\0\\0\end{pmatrix}\; e)\begin{pmatrix}1\\3\\1\\0\\0\\0\end{pmatrix}\; f)\begin{pmatrix}3\\1\\2\\0\\0\\0\end{pmatrix}$$

$$a)\begin{pmatrix}1\\2\\3\\4\\5\\6\end{pmatrix}\; b)\begin{pmatrix}3\\6\\6\\6\\6\\6\end{pmatrix}\; c)\begin{pmatrix}0\\0\\6\\6\\6\\6\end{pmatrix}\; d)\begin{pmatrix}1\\2\\3\\6\\6\\6\end{pmatrix}\; e)\begin{pmatrix}1\\4\\5\\6\\6\\6\end{pmatrix}\; f)\begin{pmatrix}3\\4\\6\\6\\6\\6\end{pmatrix}$$

Fig. 4. Top: Conditional formation of two compound orders. Bottom: Assignments to $subsum_M$ variables corresponding to order variable assignments in the Top part.

It is useful to consider a first compound (formed from the first three orders, counting down the column) and a second compound (formed from the second three). A key observation is that, counting from the top of each column, a compound can form only when enough orders have been assigned (three orders for the first compound, six for the second). To exploit this observation, for each column on which compound orders may appear, we introduce a column of variables, $subsum_M$, which record the cumulative sum of assigned orders read down the column. Figure 4 (Bottom) presents the $subsum_M$ variables for our examples.

Given the $subsum_M$ variables, we introduce a *position* variable for each compound, whose domain is the set of possible slab indices, constrained as follows:

$$\begin{aligned} subsum_M[position - 1] &< compoundSize \times instanceNo \\ subsum_M[position] &\geq compoundSize \times instanceNo \end{aligned}$$

where *compoundSize* gives the number of orders necessary to form the compound, and *instanceNo* denotes which of the compounds of *compoundSize* on this column that *position* is associated with. This pair of constraints ensure that *position* indicates a unique slab when the corresponding column of $order_M$ is assigned.

The remaining question, given some partial assignment, is whether the compound order associated with *position* has formed on the slab indicated by *position*. This is recorded in a 0/1 variable, *switch*, paired with each *position* variable and constrained as follows:

$$switch = (order_M[column][position] \geq compoundSize)$$

where *column* is the column of $order_M$ on which the compound may form.

Consider n symmetrical compound orders. We order these compounds ascending by the column on which they appear, breaking ties by ordering the 'first', 'second', ..., 'nth' compounds in a column, as defined in the previous section, ascending. We denote the *switch* and *position* variables of the ith compound under this ordering as $switch_i$ and $position_i$. The conditional symmetry can be broken straightforwardly as follows:

$$\forall i < j \in \{1, \ldots, n\} : (switch_i = 1 \wedge switch_j = 1) \rightarrow position_i \leq position_j$$

These ordering constraints are compatible with the *slabAndOrderRow* symmetry-breaking constraints given above. If the compound orders were ordered in the reverse direction, solutions might be pruned incorrectly. Given a set of symmetrical objects, it is normally only necessary to order adjacent elements in the set [4]. Here, since we cannot be certain that any particular conditional symmetry will form, we post the transitive closure of the ordering constraints.

The formation of compound order symmetry depends on the order instance data. Hence, we constructed 12 instances where compound order symmetries were highly likely to form. We used only one colour for all orders, and chose the size and number of the smaller orders so that several small orders are equivalent in size to one of the larger orders.

We ran four experiments on our test suite, summarised in the four columns of Table 1. Performance with no symmetry breaking at all was very poor, so column 1 gives results with a non-increasing ordering on $slab_M$ only (a simplification of *slabAndOrderRow* symmetry breaking). Column 2 presents the results of using full *slabAndOrderRow* symmetry breaking. Columns 3 and 4 respectively give the results of combining $slab_M$ and *slabAndOrderRow* symmetry breaking with compound order conditional symmetry breaking. The results show that the overhead of compound order symmetry breaking is significant. Although it clearly reduces search — in the instances tested a reduction of as much as 50% is gained — the time taken is increased overall

Given our results, the challenge is to make the encoding of detection of conditional symmetry of this type sufficiently lightweight that it can be used without increasing the overall search effort.

4 Breaking Conditional Symmetry by Reformulation

Modelling has a substantial effect on how efficiently a problem can be solved. An appropriate reformulation of a model can turn an insoluble problem into a soluble one in practical terms. Modelling and reformulation are equally important for symmetry breaking. Different models of the same problem can have different symmetries; one formulation can have symmetries which are easier to deal with

	$slab_M$ Symm. Breaking		$slab\&OrderRow$ Symm. Breaking		$slab_M$+ Comp. Order Symm. Breaking		$slab\&OrderRow$+Comp. Order Symm. Breaking	
Prob	Choices	Time(s)	Choices	Time(s)	Choices	Time(s)	Choices	Time(s)
1	18,014,515	1120	79,720	5.64	-	-	68,717	36.4
2	6,985,007	439	15739	1.45	-	-	13,464	6.79
3	7,721	0.741	1,798	0.26	6,461	3.48	1,472	0.971
4	155,438	8.86	60,481	4.10	49,234	31.0	30,534	16.2
5	146,076	7.48	56,590	3.45	46,599	23.4	27,921	12.4
6	117,240	6.01	49,098	2.82	39,411	17.7	24,112	9.70
7	147,148	7.1	60,035	3.34	70,881	36.3	37,672	18.0
8	171,781	8.02	77,187	4.13	80,557	37.1	45,293	19.3
9	206,138	9.52	92,526	4.87	97,072	44.9	53,666	23.0
10	348,716	16.6	140,741	7.55	178,753	94.8	84,046	41.5
11	313,840	15.7	130,524	7.21	164,615	98.5	79,621	44.4
12	266,584	13.9	110,007	6.19	138,300	82.5	68,087	37.8

Table 1. Steel Mill Slab Design: Experimental Results. Times to 3 significant figures. A dash indicates optimal solution not found within 1 hour. Hardware: PIII 750MHz, 128Mb. Software: Ilog Solver 5.3 (Windows version).

than another. Thus, reformulation of a problem can be critical in dealing with symmetries. Unfortunately, there is no general technique for suggesting reformulations for breaking symmetry. If anything, conditional symmetry intensifies the difficulties, but here we present a successful example. The all-interval series problem (problem 7 in CSPLib) is to find a permutation of the n integers from 0 to $n-1$ so that the differences between adjacent numbers are also a permutation of the numbers from 1 to $n-1$.

We can model this using n integer variables $x_0, x_1, ..., x_{n-1}$ where x_i represents the number in position i in the permutation. There is an allDifferent constraint on the x variables. Following Choi and Lee [1], we use auxiliary variables $d_i = |x_i - x_{i+1}|$ for $0 \leq i \leq n-2$ to represent the differences between adjacent numbers; these variables are required to be all different. We use lexicographic variable ordering.

There are 4 obvious symmetries in the problem: the identity, reversing the series, negating each element by subtracting it from $n-1$, and doing both. There is also a conditional symmetry: we can cycle a solution to the problem about a pivot to generate another solution. The location of this pivot is dependent on the assignments made and so these symmetries are conditional. As an example, here are two solutions for $n = 11$. Differences are written underneath the numbers:

```
0 10 1 9 2 8 3 7 4 6 5          3 7 4 6 5 0 10 1 9 2 8
 10 9 8 7 6 5 4 3 2 1            4 3 2 1 5 10 9 8 7 6
```

The difference between the first number (0) on the left and last number (5) is 5. This means we can split the sequence between the 8 and 3, losing the difference 5. We can join the rest of the sequence on to the start, because the $5 - 0$ will now replace $8 - 3$. This yields exactly the solution shown on the right. In this case the pivot is between the values 8 and 3. The difference between first and last terms must always duplicate a difference in the sequence, so this operation can be applied to any solution.

We now give a reformulation which eliminates all symmetry including conditional symmetry, with a 50-fold runtime improvement on the best previous work.

Consider a cycle formed by n nodes, with the n differences between consecutive nodes satisfying the constraint that every difference from 1 to $n-1$ appears at least once, and one difference appears exactly twice. From any solution to this we can form two all-interval series, by breaking the cycle at either one of the repeated differences. The reformulation introduces new symmetry because we can rotate the cycle, but it is broken by setting the first element to 0. Next, we note that 0 and $n-1$ must be adjacent, and since we can reverse any sequence, we insist that the second element is $n-1$. Finally, the difference $n-2$ can only appear by putting $n-2$ before 0 in the cycle, or by putting 1 after $n-1$. But after negation, reversal, and cycling, the two cases are the same. So we can insist that the sequence starts $0, n-1, 1$. This gives the reformulated problem:

Definition 1 (Reformulation of All-interval series problem). *Given $n \geq 3$, find a vector $(s_0, ..., s_{n-1})$, such that:*

1. *s is a permutation of $\{0, 1, ..., n-1\}$; and*
2. *the interval vector $(|s_1 - s_0|, |s_2 - s_1|, ...|s_{n-1} - s_{n-2}|, |s_{n-1} - s_0|)$ contains every integer in $\{1, 2, ..., n-1\}$ with exactly one integer repeated; and*
3. *$s_0 = 0$, $s_1 = n-1$, $s_2 = 1$.*

Elsewhere [10] we show that: (i) for $n > 4$, there are exactly 8 times as many solutions to the original all-interval series problem as to the reformulated one, and (ii) the repeated difference is even iff n is congruent to 0 or 1 $mod 4$. To code this formulation, we simply replaced the allDifferent constraint by a constraint to ensure that every difference occurs at least once: since there are n differences, one must automatically appear twice. Finally we added the constraint on the parity of the repeated difference, which reduced run-time by about a third.

n	Solutions	Fails	Choice Points	Cpu (sec)	Speedup	Fails/Solution
3	1	0	0	0.01	-	0
4	1	0	0	< 0.01	-	0
5	1	0	0	< 0.01	-	0
6	3	1	3	< 0.01	-	0.33
7	4	1	4	< 0.01	-	0.25
8	5	9	13	< 0.01	-	1.80
9	15	14	28	0.01	9	0.93
10	37	69	105	0.02	13	1.97
11	81	278	358	0.02	61	3.43
12	166	858	1,023	0.06	116	5.17
13	400	3,758	4,157	0.28	121	9.40
14	1,239	19,531	20,769	1.78	103	15.76
15	3,199	91,695	94,893	8.85	-	28.66
16	6,990	389,142	396,131	36.94	-	55.67
17	17,899	2,093,203	2,111,101	215.61	-	116.95
18	63,837	13,447,654	13,511,490	1,508.26	-	212.15
19	181,412	79,270,906	79,452,317	9,778.94	-	436.97
20	437,168	435,374,856	435,812,023	53,431.50	-	995.90

Table 2. Run times for reformulated version of the all-interval series problem. Where meaningful, the column for speedup indicates the factor by which these run times improve those of SBDS using the unconditional symmetries in [10] on the same machine. Our code is actually Solver 4.4 code compiled and run under Solver 5.2.

Table 2 shows results using the reformulated encoding. Where we have meaningful comparisons, from about $n = 11$ to 14, this formulation is around 100 times faster than SBDS on unconditional symmetries. This is roughly a 50-fold speedup on Puget and Régin's results [12]. It is clear that this formulation is the best way known to count solutions to the all-interval series problem. Table 2 shows that the number of fails per solution roughly doubles for each increment in n. Thus, while sometimes regarded as the easiest problem in CSPLib, the all-interval series still seems to involve considerable combinatorial search.

There is little we can say in general about reformulating to break conditional symmetry except that it can lead to dramatic performance improvements, but seems to require considerable insight on a case-by-case basis. General techniques for reformulation would be highly desirable, but remain in the future.

5 A Generic Method of Breaking Conditional Symmetries

It is preferable for breaking conditional symmetries – as it is for unconditional symmetries – to have a generic method where the symmetries and conditions can be described easily and broken efficiently. To achieve this, we look to previous methods of breaking symmetries and examine how they could be modified to cope with conditional symmetries.

Gent, McDonald and Smith [10] give two implementations of SBDS modified to work for conditional symmetries. These implementations provide proof of concept only as both have serious problems. Both methods reduce the efficiency of constraint solving. The first requires a different symmetry function for each *possible* conditional symmetry, and naturally there will always be many more than the unconditional symmetries. The second removes this problem, but the implementation is grounded heavily in the specific CSP. Thus no general purpose method proposed to date for conditional symmetries can be regarded as satisfactory. In this section we describe the main disadvantage of using SBDS-like approaches when dealing with conditional symmetry. We also explain how we have modified SBDD [3] to effectively deal with generic conditional symmetries.

5.1 The problem with using SBDS to break conditional symmetry

SBDS adds constraints to the local subtree. These constraints are discarded upon backtracking from the root node of the subtree. This means that we must have an SBDS constraint for each *possibly* applicable symmetry. This is a particularly high overhead where, as in the example of all-interval series, there are many more conditional symmetries than unconditional ones. An alternative is to check at a node whether or not a condition holds, and only to add the SBDS constraints in that local subtree where the condition is known to hold. Unfortunately, this approach fails. We might backtrack from this point and therefore discard the SBDS constraint, going back up the tree to a node where the condition is no longer true. Since the condition is not true, no conditional symmetry will be posted. Unfortunately, the condition could become true again on further reassignment of variables. Thus, this approach is untenable because it will miss duplicate solutions.

In contrast to SBDS, it seems that SBDD should adapt naturally to the conditional case. This is because the check is performed at a node about to be explored. At this point, we can calculate which conditional symmetries are known to hold. We can then calculate the resulting group, and check this against previously visited nodes. Unlike SBDS, when we backtrack from a node, we do not need to know what conditional symmetry holds in some future node. We can maintain the database of nodes visited in the same way as conventional SBDD: that is, we need merely to record the nodes at the roots of fully explored search trees. At a search node of depth d there are at most d such roots to store, which helps to make SBDD so efficient in general.

Unfortunately, the above analysis assumes that the correct algebraic structure (unconditional symmetries combined with conditional symmetries) is a group. This is not the case in general. Suppose that the unconditional symmetries form a group G, and that a conditional symmetry has group H, with both groups acting on the set of possible variable to value assignments of the CSP. If we naïvely compose all symmetries from both groups, then we may lose solutions: if the unconditional symmetry modifies the condition so that it does not hold, it is no longer sound to apply the conditional symmetry.

We now describe a sound method for breaking both conditional and unconditional symmetries dynamically using GAP–SBDD [9]. At each node in the search tree we discover each conditional symmetry that holds, and generate a symmetry group H_i for each one. We first check for dominance in G. If dominated we backtrack. If not we pick H_1 and check for dominance. Again, if dominated we backtrack, if not we pick H_2 and repeat until the H_i are exhausted.

This method is sound, since we never compose conditional and unconditional symmetries, although it introduces the computational overhead of generating groups and ensuring that the state of search maintained for each group generated. The method is incomplete in general for the same reason; it may be the case that composing symmetries results in the detection of dominance which correctly prunes the search tree. Our conditional–SBDD is the first satisfactory dynamic technique devised, and is fully generic: it can be applied to any CSP with known conditional and unconditional symmetries.

As an example we report on a prototype implementation of conditional–SBDD applied to the graceful graphs $C_3^{(t)}$ described in Section 3.1. As there, we set the central node to be 0, but no longer add constraints to break the conditional symmetries on triangles. E.g. if one node in a triangle is 12, then the numbers i and $12 - i$ are equivalent on the other node of the triangle. The results are given in Table 3. We see that conditional–SBDD results in many fewer backtracks and much less search depth, at the cost of increased GAP cpu-time taken to identify and search through the H_i. Breaking no conditional symmetries results in several isomorphic solutions being returned. Conditional–SBDD is both sound and complete for these examples – exactly one member of each class of solutions is returned.

	Conditional–SBDD						SBDD					
t	GAP cpu	ECL cpu	Σ-cpu	BT	MD	sols	GAP cpu	ECL cpu	Σ-cpu	BT	MD	sols
4	10.44	1.50	11.94	199	16	8	3.47	2.66	6.13	782	23	128
5	340.54	24.58	365.12	1823	23	21	53.21	42.44	95.65	11,255	36	672
6	8,336.74	374.73	8,711.47	18,630	31	0	907.31	954.45	1861.76	186,015	50	0

Table 3. Conditional–SBDD: Experimental Results. Times to 3 significant figures. Software: GAP v4.4 & ECLIPSE v5.7. BT denotes backtracks; MD denotes maximum depth attained during search.

6 Conclusions and Future Work

We have introduced the study of *conditional* symmetry breaking in constraint programming. We demonstrate with concrete implementations and case studies that three methods – each well-known for breaking unconditional symmetries – can be applied to conditional symmetries. These are adding conditional symmetry-breaking constraints, reformulating the problem to remove the symmetry, and augmenting the search process to break the conditional symmetry dynamically through the use of a new variant of SBDD. We can conclude that the study of conditional symmetry is as rich and fertile for new developments as unconditional symmetry breaking, It is arguably even more important in practice, since many problems contain symmetries that arise during search.

References

1. C. Choi and J.H. Lee, *On the pruning behavior of minimal combined models for CSPs.*, Proceedings of the Workshop on Reformulating Constraint Satisfaction Problems 2002.
2. J. Crawford, M. L. Ginsberg, E. Luks, A. Roy. Symmetry-breaking Predicates for Search Problems. *Proc. of the 5th KRR*, pp. 148–159, 1996.
3. T. Fahle, S. Schamberger, M. Sellmann. Symmetry breaking. In: Proc. CP 2001. (2001) 93–107
4. A. M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, T. Walsh. Global Constraints for Lexicographic Orderings. *Proc. CP 02*, pp. 93–108, 2002.
5. A. M. Frisch, I. Miguel, T. Walsh. Symmetry and Implied Constraints in the Steel Mill Slab Design Problem. *Proc. Formul '01*, pp. 8–15, 2001.
6. J. A. Gallian. Graph Labeling. *The Electronic Journal of Combinatorics*, Dynamic Surveys (DS6), `www.combinatorics.org/Surveys`,2003.
7. I.P. Gent, C.A. Jefferson, I. Lynce, I. Miguel, P. Nightingale, B.M. Smith, A. Tarim, Search in the Patience Game Black Hole CP Pod Research Report 10, 2005
8. I. P. Gent, W. Harvey, T. W. Kelsey. Groups and Constraints: Symmetry Breaking During Search *Proc. 8th CP 02*, pp. 415-430, 2002.
9. I. P. Gent, W. Harvey, T. W. Kelsey, S. A. Linton. Generic SBDD Using Computational Group Theory. *Proc. CP 03*, pp. 333-347, 2003.
10. I. P. Gent, I. McDonald, B. M. Smith. Conditional Symmetry in the All-Interval Series Problem. *Proc. SymCon 03*, 2003.
11. David Parlett. *The Penguin Book of Patience*. Penguin, 1980.
12. J.-F. Puget and J.-C. Régin, *Solving the all-interval problem*, Available from http://www.csplib.org.