A Novel Approach For Detecting Symmetries In CSP Models

B. Demoen
Katholieke Universiteit Leuven, Belgium
bmd@cs.kuleuven.ac.be

M. Garcia de la Banda, C. Mears, M. Wallace
Monash University, Australia
mbanda,cmears,wallace@infotech.monash.edu.au

Abstract

While several powerful methods exist for automatically detecting symmetries in *instances* of constraint satisfaction problems (CSPs), methods for detecting symmetries in CSP *models* are constrained to finding relatively simple symmetries, or operating on a narrow range of problems. Herein, a new approach for detecting symmetries in CSP models is presented. The approach is based on first applying powerful methods to a sequence of instances of the model, and then reasoning on the resulting instance symmetries to infer symmetries of the model. Several case studies show that this approach deserves further exploration.

1 Introduction

Constraint satisfaction problems (CSPs) can often be separated into two parts. The *model* specifies the variables, their domains, and the set of constraints that operate on the variables, but is usually parametrised by the particular number of variables, values and, thus, constraints. The *data* provides concrete values to these parameters. As a result, the model in itself represents a *class* of CSPs, while the model plus the data specifies an *instance* of that class (i.e., a particular CSP).

For example, a graph colouring model might be defined in terms of a number of variables (i.e., nodes), each coloured according to a given set of values (i.e., allowed colours), and a set of constraints indicating that no edge can join nodes of the same colour. An instance of the problem is specified by supplying a particular graph (number of nodes and edges among them) and the set of colours. Note that each instance might have a different number of variables and/or constraints.

Solving a CSP can be made more efficient by exploiting the symmetries of the problem. This is because, during search, one can omit parts of the search space that are symmetric to others already explored. If these already explored parts led to a solution, one would have avoided the time spent in searching for symmetric solutions, since they can be generated by applying the symmetries to the already found solutions. If they led to failure, one would have avoided the time spent in discovering failure yet again.

Considerable progress has been made in the automatic detection of symmetries of CSPs and their exploitation in speeding up the search (e.g., [Mears *et al.*, 2006; Cohen *et al.*, 2005; Puget, 2005; Walsh, 2006; Romani and Markov, 2005; Sellmann and Hentenryck, 2005; Mancini and Cadoli, 2005; Roney-Dougal *et al.*, 2004; Frisch *et al.*, 2003; Gent *et al.*, 2003; Puget, 2002; Gent and Smith, 2000; Haselböck, 1993] . Unfortunately, the most powerful methods [Puget, 2005; Cohen *et al.*, 2005] can only be applied to a given instance, rather than to a model. Therefore, the symmetries detected can only be used to accelerate the solving process for that instance and, as a result, the cost of detecting them cannot be amortised over all instances of the class.

Furthermore, the computation costs of these methods grow with the size of the problem instance in such a way as to render them impractical for real-size instances. For example, the most powerful and generic technique for symmetry detection [Cohen *et al.*, 2005] requires a representation of the "micro-structure" of the instance – a graph with a node for every possible value of every variable, and a hyper-edge between each set of compatible (or incompatible) nodes. Clearly, the size of this structure can grow dramatically with the size of the problem instance.

While some automatic symmetry detection methods are defined for models [Roy and Pachet, 1998; van Hentenryck *et al.*, 2005], to our knowledge, they can only detect a relatively small set of "simple" symmetries (i.e., piecewise value and piecewise variable interchangeability), and heavily depend on the precise form of the problem model (i.e., use of global constraints). Instead, we build on powerful symmetry detection techniques designed for problem instances to discover symmetries for models. This is achieved by (1) using symmetry detection methods on a series of small problem instances to elicit candidate symmetries, (2) parametrising these candidate symmetries to be defined over the model rather than over a particular instance, and (3) determining whether these are indeed problem model symmetries.

2 Background and Definitions

A CSP is a tuple (X, D, C, dom) where X represents a set of variables, D a set of domains, C a set of constraints, and where dom is a function from X to D, so that $dom(x) \in D$ denotes the domain of variable $x \in X$. By an abuse of notation, when all variables have the same domain, D will simply denote this domain and the dom-function is usually omitted.

Example 1 *Consider the* Latin square problem *of size 3, which involves a* 3×3 *square where each of the 9 elements in the square must take a value from [1..3], in such a way that each value occurs exactly once in each row and exactly once in each column. The associated CSP can be defined as*

$$X = \{x_{11}, x_{12}, x_{13}, x_{21}, x_{22}, x_{23}, x_{31}, x_{32}, x_{33}\}$$
$$D = \{1, 2, 3\}$$
$$C = \{x_{11} \neq x_{12}, x_{11} \neq x_{13}, x_{12} \neq x_{13}, x_{21} \neq x_{22}, x_{21} \neq x_{23}, x_{22} \neq x_{23}, x_{31} \neq x_{32}, x_{31} \neq x_{33}, x_{32} \neq x_{33}, x_{11} \neq x_{21}, x_{11} \neq x_{31}, x_{21} \neq x_{31}, x_{12} \neq x_{22}, x_{12} \neq x_{32}, x_{22} \neq x_{32}, x_{13} \neq x_{23}, x_{13} \neq x_{33}, x_{23} \neq x_{33}\}$$

where x_{ij} *represents the element in row* i*, column* j*.* □

For a given CSP, a *literal* lit is of the form $x = d$ where $x \in X$ and $d \in dom(x)$. We will use $var(lit)$ to denote its variable x. We denote the set of all literals of a CSP P by $lit(P)$. An *assignment* A is a set of literals. An assignment *over a set of variables* $V \subseteq X$ has exactly one literal $x = d$ for each variable $x \in V$. An assignment over X is called a *complete* assignment.

A constraint c is defined over a set of variables, denoted by $vars(c)$, and specifies a set of *allowed* assignments over $vars(c)$. An assignment over $vars(c)$ that is not allowed by c is *disallowed* by c. An assignment A over $V \subseteq X$ *satisfies* constraint c if $vars(c) \subseteq V$ and the projection of A over $vars(c)$ (i.e., $\{lit \in A | var(lit) \in vars(c)\}$), is allowed by c. A *solution* is a complete assignment that satisfies every constraint in C.

A *solution symmetry* f for a CSP P is a permutation of $lit(P)$ that preserves the set of solutions [Cohen *et al.*, 2005], i.e., a bijection from literals to literals that maps solutions to solutions. A *constraint symmetry* is a solution symmetry that preserves the set of constraints. Two important kinds of solution symmetries are induced by either permuting variables or values.

A permutation f of the set X of variables induces a permutation p_f of literals by defining $p_f(x = d)$ as the literal $f(x) = d$. A *variable symmetry* is a permutation of the variables whose induced literal permutation is a solution symmetry [Puget, 2002]. Since the inverse of any such permutation is also a symmetry, we will use $\langle x_1, x_2, \ldots, x_n \rangle \leftrightarrow \langle x_{1'}, x_{2'}, \ldots, x_{n'} \rangle$, where $\{x_1, \ldots, x_n\}, \{x_{1'}, \ldots, x_{n'}\} \subset X$ to denote the symmetry which maps each x_i to $x_{i'}$ leaving the remaining variables in X unchanged.

A set of domain permutations $f_{dom(x)}$, one for each $x \in X$, induces a permutation p_f of literals by defining $p_f(x = v)$ as the literal $x = f_{dom(x)}(v)$. A *value symmetry* is a set of domain permutations whose induced literal permutation is a solution symmetry [Puget, 2002]. We will use $\langle d_{i1}, d_{i2}, \ldots, d_{in} \rangle \leftrightarrow \langle d_{i1'}, d_{i2'}, \ldots, d_{in'} \rangle$, where $\{d_{i1}, d_{i2}, \ldots, d_{in}\} = dom(x_i) = \{d_{i1'}, d_{i2'}, \ldots, d_{in'}\}$, to denote a value symmetry for $x_i \in X$. A *variable-value* symmetry is any solution symmetry that is not a variable or a value symmetry. Note that it is not necessarily a composition of a variable and a value symmetry.

Example 2 *The problem of Example 1 has:*

- *variable symmetries that swap any two columns:* $\langle x_{11}, x_{21}, x_{31} \rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32} \rangle$, $\langle x_{11}, x_{21}, x_{31} \rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33} \rangle$, *and* $\langle x_{12}, x_{22}, x_{32} \rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33} \rangle$.
- *similar variable symmetries that swap any two rows.*
- *variable-value symmetries that transpose the rows, column and value dimensions, and correspond to flipping the* 3×3 *square using a diagonal.*□

Several methods [Romani and Markov, 2005; Puget, 2005; Cohen *et al.*, 2005] have been proposed to automatically detect the symmetries of a CSP instance by constructing a (hyper-)graph representation of the CSP instance, and using graph automorphism techniques to detect symmetries. Our approach uses the technique of Mears et al. [Mears *et al.*, 2006] since it is more powerful than that of Puget [Puget, 2005] without being as computationally demanding as that of Cohen et al.[Cohen *et al.*, 2005]. However, any such method can be used. The idea is to (a) represent every literal as a node, (b) represent every assignment disallowed by a constraint as a hyper-edge, and (c) add an edge between every two literals $x = d_1$ and $x = d_2$ where $d_1 \neq d_2$.

Example 3 *Consider the CSP provided in Example 1. The associated graph (left hand side of Figure 1) has 9×3=27 nodes (labelled* $\binom{[i,j]}{k}$ *) representing the 27 literals* $x_{i,j} = k$ *where* $i, j, k \in [1..3]$*, and (18*3) + (9*3) edges representing the 3 assignments disallowed by each of the 18 constraints, and the 3 extra edges needed to disallow each pair of values of the 9 variables.*□

Given a hyper-graph $\langle V, E \rangle$, where V is a set of nodes, and E a set of unweighted and undirected hyper-edges, an *automorphism* f of graph $\langle V, E \rangle$ is a permutation of the nodes (i.e., a bijection among nodes) such that $\forall \{n_i, \cdots, n_j\} \in E : \{f(n_i), \cdots, f(n_j)\} \in E$. For a CSP problem P the graph has a node for each literal in $lit(P)$. Since a graph automorphism is a permutation of the nodes of the graph, it has a direct interpretation as a permutation of the literals in $lit(P)$. In particular, each graph automorphism corresponds to a symmetry of P. Thus, in an abuse of terminology, we will sometimes use *symmetry of a graph* as a shorthand for *automorphism of the graph* associated to a CSP.

Standard tools, such as Saucy [Darga *et al.*, 2004], can compute the automorphisms of a graph instance, and return the resulting symmetry group (i.e., all possible symmetries) by means of a set of generators (i.e., a possibly minimal set of symmetries that can be used to generate all other elements).

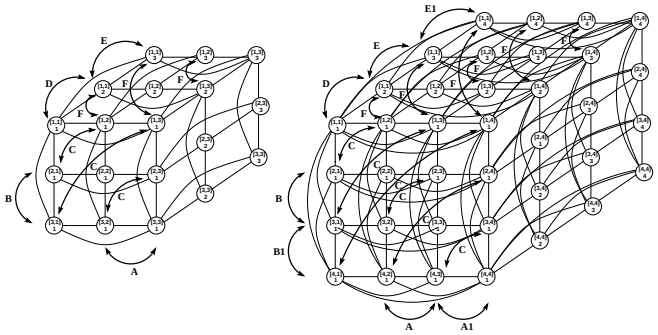


Figure 1: Graphs and generators for LatinSquare[3] and LatinSquare[4]

Example 4 *For the graph of the Latin square problem of size 3 given in Example 3 Saucy returns the following (non-minimal) set of generators:*

$$\mathbf{A}\ \langle n_{121}, n_{122}, n_{123}, n_{221}, n_{222}, n_{223}, n_{321}, n_{322}, n_{323}\rangle \leftrightarrow \langle n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}, n_{331}, n_{332}, n_{333}\rangle$$
$$\mathbf{B}\ \langle n_{211}, n_{212}, n_{213}, n_{221}, n_{222}, n_{223}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow \langle n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}, n_{331}, n_{332}, n_{333}\rangle$$
$$\mathbf{C}\ \langle n_{121}, n_{122}, n_{123}, n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow \langle n_{211}, n_{212}, n_{213}, n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}\rangle$$
$$\mathbf{D}\ \langle n_{111}, n_{121}, n_{131}, n_{211}, n_{221}, n_{231}, n_{311}, n_{321}, n_{331}\rangle \leftrightarrow \langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{332}\rangle$$
$$\mathbf{E}\ \langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{333}\rangle \leftrightarrow \langle n_{113}, n_{123}, n_{133}, n_{213}, n_{223}, n_{233}, n_{313}, n_{323}, n_{333}\rangle$$
$$\mathbf{F}\ \langle n_{112}, n_{113}, n_{123}, n_{212}, n_{213}, n_{223}, n_{312}, n_{313}, n_{323}\rangle \leftrightarrow \langle n_{121}, n_{131}, n_{132}, n_{221}, n_{231}, n_{232}, n_{321}, n_{331}, n_{332}\rangle$$

where, for reasons of space, node n_{ijk} represents literal $x_{i,j} = k$. The meaning of these generators, illustrated in the left hand side of Figure 1, is as follows: **A** *indicates that column 2 can be swapped with column 3,* **B** *that row 2 can be swapped with row 3,* **C** *that the square can be reflected across the top-left/bottom-right diagonal,* **D** *that value 1 can be swapped with value 2,* **E** *that value 2 can be swapped with value 3, and* **F** *that the second dimension of the square can be swapped with the value dimension.*

The combination of these generators results in the symmetries given in Example 2. For example, swapping columns 1 and 2 ($\langle x_{11}, x_{21}, x_{31}\rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32}\rangle$) can be achieved by first applying **F**, *then* **D**, *and then* **F**.□

Note that, in order to be able to use these automorphism tools, we need to convert each hyper-edge into a set of binary edges. This can easily be done [Puget, 2005] by (a) creating a new kind of node (a *constraint* node) with an edge to every literal in the disallowed assignment, (b) assigning a different colour to each kind of node (e.g., black to constraint nodes and white to the rest), and (c) extending the concept of automorphism to ensure that the colour of a node is preserved.

3 Parametrising the CSP and its associated graph

There is no standard notation for distinguishing between a CSP model and a CSP instance. Herein, we shall denote a CSP model as CSP[Data], where *Data* represents the parameters to the model, and the instance as CSP[d], where d represents their particular values.

For simplicity, herein we will use mathematical notation to represent CSP models. However, any high-level modelling language, such as ESO [Mancini and Cadoli, 2005], OPL [Hentenryck, 1999], Essence [Frisch *et al.*, 2007], Esra [Flener *et al.*, 2004], and Zinc [de la Banda *et al.*, 2006], can be used, as long as it explicitly separates the model from the data, has multi-dimensional arrays of finite domain variables, and supports iteration over these arrays.

Example 5 *The Latin square problem of Example 1 can be parametrised on the size N of the board as LatinSquare[N], and can be modelled as:*

$$X[N] = \{square_{ij} | i, j \in [1..N]\}$$
$$D[N] = [1..N]$$
$$C[N] = \{square_{ij} \neq square_{ik} | i, j \in [1..N], k \in [j+1..N]\} \cup \{square_{ji} \neq square_{ki} | i, j \in [1..N], k \in [j+1..N]\}$$

which defines $N \times N$ integer decision variables ($square_{ij}$) with values in $[1..N]$, and conjoins the inequality constraints for every row (`i`*) and column (*`j`*).* □

While being able to obtain the graph associated to an instance is useful, our aim is to determine the symmetries for the model. Thus, we are interested in obtaining a graph that can capture all instances of the model, i.e., a *parametrised graph* that, when instantiated for a given value, yields the graph associated to the problem instance. Note that, the parametrised graph is simply a syntactic construct that represents a class of graphs, much as the parametrised model represents a class of instances. We denote by $G[Data]$ the parametrised graph obtained from model $CSP[Data]$, and by $G[d]$ the graph of instance $CSP[d]$. Furthermore, we would like the graph specification to capture some of the knowledge about the structure of the parametrised problem.

Formally, the parametrised graph $G[Data]$ obtained for model $CSP[Data] = (X[Data], D[Data], C[Data])$ can be obtained as follows:

- $G[Data] = \langle V, E_v \cup E_c\rangle$
- $V = \{x_i = d_i | x_i \in X[Data], d_i \in dom(x_i)[Data]\}$, i.e., V contains a node for every literal in the model.
- $E_v = \{\{x = d_i, x = d_j\} | x \in X[Data], d_i, d_j \in dom(x)[Data], i \neq j\}$, i.e., an edge exists between every two nodes that map a variable to different values.
- $E_c = \bigcup_{c \in C[Data]} \{A | var(A) = var(c)$, *$A$ is an assignment disallowed by c*$\}$, i.e., a hyper-edge exists for every disallowed assignment A of every constraint c, and connects the nodes associated to all literals in A.

Example 6 *The parametrised graph $G[N]$ associated to the LatinSquare[N] model of Example 5 is computed as follows. The set of literals that can be extracted from the model is $\{square_{ij} = v | i, j, v \in [1..N]\}$. This yields the associated set of nodes $\{n_{ijv} | i, j, v \in [1..N]\}$, in the parametrised graph. Note that the nodes in $G[N]$ maintain some of the knowledge about the structure of LatinSquare[N] thanks to the reuse of the i and j identifiers. This is important not only to automate the construction of the edges in $G[N]$, but as we will see later, to parametrise symmetries belonging to graph instances.*

E_v is defined as $\{\{n_{ijv_1}, n_{ijv_2}\} | i, j, v_1, v_2 \in [1..N], v_1 \neq v_2\}$, while E_c is obtained by transforming the two constraints in the model into the set of assignments they disallow. If $c \in C[N]$ and $vars(c) = \langle square_{i_1 j_1}, square_{i_2 j_2} \rangle$, then there is an edge $\{n_{i_1 j_1 v_1}, n_{i_2 j_2 v_2}\}$ for each v_1, v_2 such that $\langle square_{i_1 j_1} = v_1, square_{i_2 j_2} = v_2 \rangle$ is disallowed by c. Formally:

$$E_c = \{\{n_{ijv}, n_{ikv}\} | i, j, v \in [1..N], k \in [j+1..N]\} \cup \{\{n_{jiv}, n_{kiv}\} | i, j, v \in [1..N], k \in [j+1..N]\} \square$$

Given a parametrised graph, it is also possible to express a *parametrised permutation* of its nodes. For example, the following is a parametrised permutation f of $G[N]$ for LatinSquare[N]: $f(n_{ijv}) = n_{jiv}, \forall i, j, v \in [1..N]$. If a parametrised permutation is an automorphism of the parametrised graph for all possible parameter values, then we call it a symmetry of the parametrised graph. Thus, we can use S_{Data} to denote the group of symmetries of the parametrised graph $G[Data]$ associated to model CSP[Data]

4 Computing the candidate symmetries for a model

The idea is to compute the symmetries of several small instances of the model, and use them to elicit parametrised permutations that are likely to be symmetries of the model itself. This poses two main challenges: (a) to find automorphisms $f[d1]$, $f[d2]$ etc. of small graphs $G[d1]$, $G[d2]$ etc. that are (likely to be) instances of a single symmetry $f[Data]$ of the parametrised graph $G[Data]$, and (b) to compute from several such permutations, a parametrised permutation of which they are all instances. Both challenges are explored in this section.

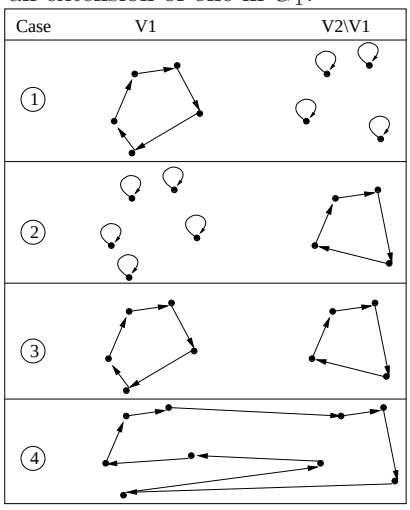
4.1 Models ordered by the subgraph relationship

When instantiated, a parametrised permutation of the parametrised graph $G[Data]$ becomes a permutation $f[d1]$ of graph $G[d1]$ and $f[d2]$ of graph $G[d2]$. This and the following subsections explore the relationship between $f[d1]$ and $f[d2]$.

We will only consider a limited, but important class of *ordered* models where graphs of different instances can be ordered by a subgraph relationship. Typically, ordered models are parametrised by a sequence of integers, one for each independent "dimension" of the model (naturally, if more than one dimension exists, the ordering is partial). For example, LatinSquare[N] is ordered since, if $n < m$, $G[n]$ is a subgraph of $G[m]$.

Let $G_1 = \langle V_1, E_1 \rangle$ and $G_2 = \langle V_2, E_2 \rangle$ be two graphs and let $G_1 \subset G_2$ indicate that G_1 is a subgraph of G_2. Automorphism f of G_2 can be *restricted* to $G_1 \subset G_2$, denoted by $f|_{G_1}$, if $\forall n \in V_1, f(n) \in V_1$, i.e., if it maps G_1 onto itself. Let G_1 and G_2 be two instances of parametrised graph $G[Data]$, and let S_1 and S_2 denote the group of automorphisms on G_1 and G_2, respectively. If $G_1 \subset G_2$, S_2 can be partitioned into two sets: the set Old_{12} of automorphisms that can be restricted to G_1, and the set New_{12} of automorphisms that cannot. Intuitively, Old_{12} contains the automorphisms in G_1 that still exist in G_2 (since they map nodes of G_1 to G_1 and of G_2 to G_2), while New_{12} contains those that have emerged for G_2 (i.e., map some nodes of G_1 to G_2 and vice versa).

We found it useful to visually illustrate the elements of S_2 according to the adjoining Figure, where cases 1, 2 and 3 belong to Old_{12}, while 4 belongs to New_{12}. Case 1 corresponds to a non-trivial permutation of nodes in G_1, extended with the identity on the new nodes of G_2. Case 2 corresponds to the trivial identity permutation of nodes in G_1, extended with a non-trivial permutation on the new nodes of G_2. Case 3 corresponds to a non-trivial permutation of nodes in G_1, extended with a non-trivial permutation on the new nodes of G_2. And case 4 corresponds to a non-trivial permutation of nodes in G_2 that is not an extension of one in G_1.



Let $D = V_2 \setminus V_1$ be the set of new nodes in G_2. Candidate symmetries belonging to cases 1, 2 and 3 above are efficiently computed in our approach by using GAP [GAP, 2006] to find the intersection between the group S_2 and the group $\{s_1 \times p | s_1 \in S_1, p \in Perm(D)\}$, where $Perm(D)$ denotes the set of all possible permutations among the nodes in D. Candidate symmetries for case 4 are found using a different approach outlined in Section 4.3 below.

Example 7 *Consider the graph $G[4]$ associated to LatinSquare[4], shown in the right hand side of Figure 1. Saucy finds 9 generators for this graph. Six of them are simple extensions of the generators found for LatinSquare[3] in Example 4. For example, the extension of*

generator **A** *is:*

$\mathbf{A}\ \langle n_{121}, n_{122}, n_{123}, n_{124}, n_{221}, n_{222}, \ldots, n_{321}, \ldots, n_{421}, \ldots\rangle \leftrightarrow$
$\langle n_{131}, n_{132}, n_{133}, n_{134}, n_{231}, n_{232}, \ldots, n_{331}, \ldots, n_{431}, \ldots\rangle$

and similarly for **B, C, D, E** *and* **F**. *The other three generators found are:*

$\mathbf{A}\ \langle n_{131}, n_{132}, n_{133}, n_{134}, n_{231}, n_{232}, \ldots, n_{331}, \ldots, n_{431}, \ldots\rangle \leftrightarrow$
$\langle n_{141}, n_{142}, n_{143}, n_{144}, n_{241}, n_{242}, \ldots, n_{341}, \ldots, n_{441}, \ldots\rangle$
$\mathbf{B1}\langle n_{311}, n_{312}, n_{313}, n_{314}, n_{321}, n_{322}, \ldots, n_{331}, \ldots, n_{341}, \ldots\rangle \leftrightarrow$
$\langle n_{411}, n_{412}, n_{413}, n_{414}, n_{421}, n_{422}, \ldots, n_{431}, \ldots, n_{441}, \ldots\rangle$
$\mathbf{E1}\langle n_{113}, n_{123}, n_{133}, n_{143}, n_{213}, n_{223}, \ldots, n_{313}, \ldots, n_{413}, \ldots\rangle \leftrightarrow$
$\langle n_{114}, n_{124}, n_{134}, n_{144}, n_{214}, n_{224}, \ldots, n_{314}, \ldots, n_{414}, \ldots\rangle$

The generators found by GAP to be in the Old_{34} *partition of the symmetry group in* $G[4]$ *are* **A, B, C, D, E** *and* **F**. *Note that all these generators correspond to case 3, while* **A1, B1** *and* **E1** *correspond to case 4.*□

4.2 Eliciting Parametrised Permutations

In order for our implementation to be able to elicit a parametrised permutation from a candidate permutation, every node in the graph must be identified by a sequence $i, j, k, \ldots$ of indices which can take any value between 1 and the dimension of the instance (such as nodes n_{ijk} in the LatinSquare[n] instance graph). If so, our implementation currently attempts to "lift" every index by identifying one of two simple situations: (I) an index which is always mapped to itself, and (II) an index which is always mapped to another. While obviously incomplete, these two heuristics alone suffice to handle all case 3 type symmetries for LatinSquare. Note that our method does not claim to elicit all possible symmetries. This is not only due to the limits of our heuristics but also due to the limits of the graph representation which only captures constraint symmetries and, thus, might miss some solution symmetries.

Example 8 *The generators found by GAP to be in* Old_{34} *for LatinSquare[3] and LatinSquare[4] in Example 7 can be automatically parametrised as:*

$\mathbf{A}\ \{n_{i2v} \leftrightarrow n_{i3v} | i, v \in [1..N]\}$
$\mathbf{B}\ \{n_{2jv} \leftrightarrow n_{3jv} | j, v \in [1..N]\}$
$\mathbf{C}\ \{n_{ijv} \leftrightarrow n_{jiv} | i, j, v \in [1..N]\}$
$\mathbf{D}\ \{n_{ij1} \leftrightarrow n_{ij2} | i, j \in [1..N]\}$
$\mathbf{E}\ \{n_{ij2} \leftrightarrow n_{ij3} | i, j \in [1..N]\}$
$\mathbf{F}\ \{n_{ijv} \leftrightarrow n_{ivj} | i, j, v \in [1..N]\}$

where **A, B, D** *and* **E** *used (I) above, while* **C** *and* **F** *used (I) and (II). For example,* **A** *is parametrised as* $\{n_{i21} \leftrightarrow n_{i31}, n_{i22} \leftrightarrow n_{i32}, n_{i23} \leftrightarrow n_{i33} | i \in [1..3]\}$ *first, then to* $\{n_{i2v} \leftrightarrow n_{i3v} | i, v \in [1..3]\}$, *and finally to the parametrised form above by replacing* $[1..3]$ *by a range that is independent of the instance. Similarly,* **C** *is parametrised to* $\{n_{ij1} \leftrightarrow n_{ji1}, n_{ij2} \leftrightarrow n_{ji2}, n_{ij3} \leftrightarrow n_{ji3} | i, j \in [1..3]\}$, *then to* $\{n_{ijk} \leftrightarrow n_{jik} | i, j, k \in [1..3]\}$, *and finally to the one shown above.*

Note that the parametrised version of these generators is identical to that obtained using the generators in LatinSquare[4]. One could then argue that there is no need to compute Old_{34} *by calling GAP, since one could simply parametrise the generators for the two instances and then check which ones are identical, thus obtaining parametrised generators known to be in* S_4. *While this works for LatinSquare[N], it is too weak in general because it depends crucially on the set of generators chosen for representing the groups. Instead, GAP intersects the groups specified by the generators and is thus independent of the particular choice of generators.* □

As one can see in Example 8, some parametrised generators contain concrete numbers as node identifiers. We take this as an indication of the possibility of further parametrising the generators in Old_{34}. However, these further parametrisations often require us to consider more than two instances of the graph.

Example 9 *The generators found by Saucy for instance LatinSquare[5] are the simple extensions of* **A, A1, B, B1, C, D, E, E1** *and* **F**, *plus three more, which we will call* **A2, B2, and E2**. *Again, all generators in instance LatinSquare[4] are found to be in* Old_{45}. *They can be automatically parametrised as:*

$\mathbf{A1}\{n_{i3v} \leftrightarrow n_{i4v} | i, v \in [1..N]\}$
$\mathbf{B1}\{n_{3jv} \leftrightarrow n_{4jv} | j, v \in [1..N]\}$
$\mathbf{E1}\{n_{ij3} \leftrightarrow n_{ij4} | i, j \in [1..N]\}$

This is a pattern that can be observed when intersecting any two consecutive instances N *and* $N+1$*: all generators in* N *are found to be in* $Old_{N(N+1)}$ *while the number of generators for* $N+1$ *increases by exactly 3, one per dimension. For example, those for LatinSquare[5] are:*

$\mathbf{A2}\{n_{i4v} \leftrightarrow n_{i5v} | i, v \in [1..N]\}$
$\mathbf{B2}\{n_{4jv} \leftrightarrow n_{5jv} | j, v \in [1..N]\}$
$\mathbf{E2}\{n_{ij4} \leftrightarrow n_{ij5} | i, j \in [1..N]\}$ □

When the number of generators in *Old* differs for two different pairs of instances, one must consider which generators to mark as candidates. One possibility is to choose the smaller set in the hope of minimising false candidates. Another is to choose the bigger set in the hope of maximising true candidates. A third is to look for patterns among the generators of the different *Old*s.

Example 10 *Each of the three generators obtained outside the Old of each instance belongs to a sequence: sequence* **A:A1:A2**, *sequence* **B:B1:B2**, *and sequence* **D:E:E1:E2**, *all starting with generators that belong to the current Old (*__A, B__ *and* __D__ *for LatinSquare[5]), and finishing with generators that do not (*__A2, B2__ *and* __E2__, *respectively). Furthermore, each sequence can itself be parametrised resulting in the following candidates:*

$\mathbf{A}\ \{\{n_{ijv} \leftrightarrow n_{ikv} | i, v \in [1..N]\} | j \in [1..N-1], k = j+1\}$
$\mathbf{B}\ \{\{n_{ijv} \leftrightarrow n_{kjv} | j, v \in [1..N]\} | i \in [1..N-1], k = v+1\}$
$\mathbf{C}\ \{n_{ijv} \leftrightarrow n_{jiv} | i, j, v \in [1..N]\}$
$\mathbf{D}\ \{\{n_{ijv} \leftrightarrow n_{ijw} | i, j \in [1..N]\} | v \in [1..N-1], w = v+1\}$
$\mathbf{F}\ \{n_{ijv} \leftrightarrow n_{ivj} | i, j, v \in [1..N]\}$ □

Note that none of these parametrised generators contain concrete numbers. Also note that, as indicated later, such sophisticated parametrisations are beyond the capabilities of our current implementation.

4.3 Using New to determine other likely candidates

Up to now we have only used Old to determine candidates. However, it is possible for an automorphism in New to be a likely candidate. To decide whether this is the case, the last step in our search for likely candidates is to parametrise every generator not in Old and mark as likely candidates those parametrisations that represent generators of different instances.

Example 11 *The* queens *problem aims at positioning N queens in an $N \times N$ chess board without one queen attacking another. The following model of Queens[N] uses N integer variables (each representing the row in which the queen is positioned) with domains in [1..N].*

$$X[N] = \{q_i | i \in [1..N]\}$$
$$D[N] = [1..N]$$
$$C[N] = \{q_i \neq q_j | i \in [1..N], j \in [i+1..N]\} \cup \{q_i + i \neq q_j + j | i \in [1..N], j \in [j+1..N]\} \cup \{q_i - i \neq q_j - j | i \in [1..N], j \in [j+1..N]\}$$

Its parametrised graph $G[N] = (V, E_c \cup E_v)$ is:

$$V = \{q_{iv} | i, v \in [1..N]\}$$
$$E_c = \{\{q_{iv}, q_{jv}\} | i, v \in [1..N], j \in [i+1..N])\} \cup \{\{q_{iv_i}, q_{jv_j}\} | i, v_i, v_j \in [1..N], j \in [i+1..N], v_i + i = v_j + j)\} \cup \{\{q_{iv_i}, q_{jv_j}\} | i, v_i, v_j \in [1..N], j \in [i+1..N], v_i - i = v_j - j)\}$$
$$E_v = \{\{q_{iv_i}, q_{jv_j}\} | i, v_i, v_j \in [1..N], v_i \neq v_j\}$$

where node q_{iv} represents literal $q_i = v$. Figure 2 shows the graph instances $G[4]$ and $G[5]$ together with the generators found by Saucy for $G[4]$:

$$\mathbf{A}\ \langle q_{11}, q_{12}, q_{21}, q_{22}, q_{31}, q_{32}, q_{41}, q_{42} \rangle \leftrightarrow \langle q_{14}, q_{13}, q_{24}, q_{23}, q_{34}, q_{33}, q_{44}, q_{43} \rangle$$
$$\mathbf{B}\ \langle q_{12}, q_{13}, q_{14}, q_{23}, q_{24}, q_{34} \rangle \leftrightarrow \langle q_{21}, q_{31}, q_{41}, q_{32}, q_{42}, q_{43} \rangle$$

and for $G[5]$:

$$\mathbf{A1}\ \langle q_{11}, q_{12}, q_{21}, q_{22}, q_{31}, q_{32}, q_{41}, q_{42}, q_{51}, q_{52} \rangle \leftrightarrow \langle q_{15}, q_{14}, q_{25}, q_{24}, q_{35}, q_{34}, q_{45}, q_{44}, q_{55}, q_{54} \rangle$$
$$\mathbf{B}\ \langle q_{12}, q_{13}, q_{14}, q_{15}, q_{23}, q_{24}, q_{25}, q_{34}, q_{35}, q_{54} \rangle \leftrightarrow \langle q_{21}, q_{31}, q_{41}, q_{41}, q_{32}, q_{42}, q_{52}, q_{43}, q_{53}, q_{45} \rangle$$

where **B** *is an extension of the generator with the same name found for $G[4]$, and* **A1** *is a new generator. As represented visually in Figure 2, generators* **A** *and* **A1** *indicate a reflection around a horizontal axis through the centre of the board, while* **B** *indicates a reflection around the top-left/bottom-right diagonal. As a group, they provide all the symmetries of a square. The generators found for $G[6]$ are, again, an extension of* **B** *and a new generator* **A2** *which also reflects the board through its horizontal axis.*

Generator **B** *is the only generator found by GAP to be in Old_{45} and also the only one in Old_{56}. Its automatic parametrisation results in $\{q_{ij} \leftrightarrow q_{ji} | i, j \in [1..N]\}$. Since (a) the number of generators in Old for $G[4]$ and $G[5]$, and for $G[5]$ and $G[6]$ is the same, and (b) its parametrised version contains no concrete numbers,* **B** *can be marked as a likely candidate.*

To decide whether **A, A1** *and* **A2** *are likely candidates or not, we parametrise them and check whether their parametrised version is identical. Our current implementation parametrises* **A** *to $\{\langle q_{i1}, q_{i2} \rangle \leftrightarrow \langle q_{i4}, q_{i3} \rangle | i \in [1..N]\}$,* **A1** *to $\{\langle q_{i1}, q_{i2} \rangle \leftrightarrow \langle q_{i5}, q_{i4} \rangle | i \in [1..N]\}$, and* **A2** *to $\{\langle q_{i1}, q_{i2}, q_{i3} \rangle \leftrightarrow \langle q_{i6}, q_{i5}, q_{i4} \rangle | i \in [1..N]\}$. Though* **A, A1** *and* **A2** *do not form a sequence such as that in Example 10, a more sophisticated parametrisation would capture the common pattern $\{q_{ij} \leftrightarrow q_{i(N-j+1)} | i, j \in [1..N]\}$ and, thus, that their parametrised version should be marked as likely candidate.*□

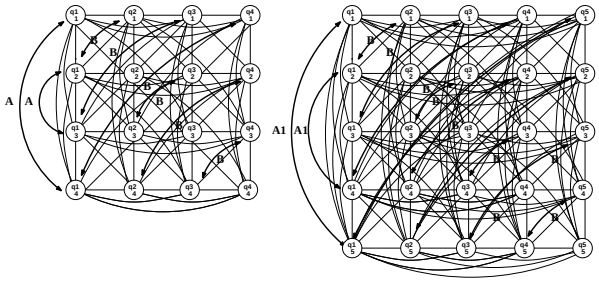


Figure 2: Graph instances for Queens[4] and Queens[5]

5 Summary of the approach

In summary, given a parametrised CSP model $CSP[Data]$, our approach:

1. obtains the associated parametrised graph $G[Data]$,
2. instantiates $G[Data]$ with at least three consecutive values $G[d_1], G[d_2]$ and $G[d_3]$, for each possible dimension of $Data$,
3. computes Old_{12} for $G[d_1]$ and $G[d_2]$, and Old_{23} for $G[d_2]$ and $G[d_3]$
4. independently parametrises each generator in Old_{12} and Old_{23},
   (a) If the number of generators in Old_{12} and Old_{23} is the same, it checks whether they result in the same set. If so, it marks each parametrised generator in either Old_{12} or Old_{23} as likely candidates. If not (often because of the occurrence of concrete values), it attempts to discover more complex patterns that eliminate these concrete values to yield common parametrisations. If it succeeds, it marks them as likely candidates.
   (b) If the number of generators is not the same, it attempts to discover sequences that eliminate concrete values to yield common parametrisations. Again, it marks them as likely candidates.
5. computes $New_i, i \in [1..3]$ as the set of generators in $G[d_i]$ and not in $Old_{12} \cup Old_{23}$. It parametrises these three sets of generators and determines likely candidates in the same way as before.
6. determines, for every likely candidate, whether it is a model symmetry.

The last step can be achieved by representing both the CSP model and the candidate in the logic formalism described in [Mancini and Cadoli, 2005], and then making use of theorem proving techniques. Of course, such

technique is in general undecidable. Another approach, which we are currently exploring, is to use graph techniques to prove that a likely candidate is an automorphism of the parametrised graph G[Data].

This approach has not completely been implemented yet. Currently, our implementation takes several user-specified ECL^{i}PS^e instances (rather than a model as in step 2 above), obtains their associated graphs, and computes their symmetry groups. Then, for each two groups of symmetries, we automatically compute *Old* (step 3 above) by calling GAP, as explained in Section 4.1. Each of the generators in the computed *Old* is then automatically parametrised (step 4) by the simple "pattern recognition" program introduced in Section 4.2. The same process is followed for each generator in each New_i (step 5), as explained in Section 4.3. We are currently exploring an automatic way of increasing the number of patterns recognised in steps 4 and 5 by using data-mining.

Let us now further illustrate our approach by means of two detailed examples.

Golomb ruler: a set of N integers (marks on the ruler) $0 = a_1 < a_2 < ... < a_N$ such that the $\frac{N(N-1)}{2}$ differences $a_j - a_i, 1 \leq i < j \leq N$ are distinct. The problem involves finding a valid set of N marks. The following model for Golomb[N] uses N integer variables (the marks) with domains in $[0..N^2]$, plus $\frac{N(N-1)}{2}$ integer variables (the differences) with domains $[0..N^2]$.

$$\begin{array}{ll} X[N] = & \{mark_i | i \in [0..N]\} \cup \{diff_{ij} | i \in [1..N], j \in [i+1..N]\} \\ D[N] = & [1..N*N] \\ C[N] = & \{mark_i - mark_j = diff_{ij} | i \in [1..N], j \in [i+1..N]\} \cup \\ & \{diff_{ij} \neq diff_{ik} | i, j \in [1..N], k \in [j+1..N]\} \end{array}$$

The parametrised graph associated to Golomb[N] is:

$$\begin{array}{ll} V = & \{m_{iv} | i \in [1..N], v \in [1..N^2]\} \cup \\ & \{d_{ijv} | i \in [1..N], j \in [(i+1)..N], v \in [1..N^2]\} \\ E_c = & \{\{m_{iv_1}, m_{jv_2}, d_{ijv_3}\} | i \in [1..N], j \in [(i+1)..N], \\ & v_1, v_2, v_3 \in [1..N^2], v_1 - v_2 \neq v_3\} \cup \\ & \{\{d_{ijv}, d_{ijv}\} | i \in [1..N], j \in [(i+1)..N], v \in [1..N^2]\} \\ E_v = & \{(m_{iv_1}, m_{iv_2}) | i \in [1..N], v_1, v_2 \in [1..N^2], v_1 \neq v_2\} \cup \\ & \{(d_{ijv_1}, d_{ijv_2}) | i \in [1..N], j \in [(i+1)..N], v_1, v_2 \in [1..N^2], \\ & v_1 \neq v_2\} \end{array}$$

where node m_{iv} represents literal $mark_i = k$ and node d_{ijv} literal $diffs_{ij} = v$. The generator for $G[3]$ is:

A $\langle d_{121}, d_{122}, d_{123}, d_{124}, d_{125}, d_{126}, d_{127}, d_{128}, d_{129} \rangle \leftrightarrow \langle d_{231}, d_{232}, d_{233}, d_{234}, d_{235}, d_{236}, d_{237}, d_{238}, d_{239} \rangle$ plus $\langle m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}, m_{16}, m_{17}, \ldots, m_{24} \rangle \leftrightarrow \langle m_{39}, m_{38}, m_{37}, m_{36}, m_{35}, m_{34}, m_{33}, m_{32}, \ldots, m_{25} \rangle$

which swaps the lengths of the spaces between the marks, i.e., turns the ruler back-to-front. The generators **A1** for $G[4]$ and **A2** for $G[5]$ follow a similar pattern. While both Old_{34} and Old_{45} are empty, $New_1 = \{\mathbf{A}\}, New_2 = \{\mathbf{A_1}\}$ and $New_3 = \{\mathbf{A_2}\}$. Their independent parametrisation results in three sets with the same number of elements but different parametrisations due to the existence of concrete values. For example, **A** is parametrised to $\{d_{12v} \leftrightarrow d_{23v} | v \in [1..N^2]\} \cup \{\langle m_{1v_1}, m_{2v_2} \rangle \leftrightarrow \langle m_{3v'_1}, m_{2v'_2} \rangle | v_1, v_2, v'_1, v'_2 \in [1..N^2], v_1 = N^2 - v'_1 + 1, v_2 = N^2 - v'_2 + 1\}$. All this has been achieved by our implementation. A more sophisticated parametrisation would further realise that the number of elements in the three sets is the same, and therefore find the pattern: $\{d_{ijv} \leftrightarrow d_{klv} | i, j, k, l \in [1..N], i = N - k, j = N - l + 2, v \in [1..N^2]\} \cup \{m_{iv_1} \leftrightarrow m_{jv'_1} | i, j \in [1..N], i = N - j + 1, v_1, v'_1 \in [1..N^2], v_1 = N^2 - v'_1 + 1\}$, and mark it as likely candidate.

Social Golfers: aims at constructing a schedule for a band of golfers that, each week, is partitioned into evenly-sized groups. Each pair of golfers may play in the same group at most once. The problem asks for a schedule of W weeks, with G groups per week and P players per group.

$$\begin{array}{ll} X[N] = & \{players_{wg} | w \in [1..W], g \in [1..G]\} \\ D[N] = & \wp(\{1..P*G\}) \\ C[N] = & \{|players_{wg}| = P | w \in [1..W], g \in [1..G]\} \cup \\ & \{|players_{wg_1} \cap players_{wg_2}| = 0 | w \in [1..W], \\ & \quad g_1, g_2 \in [1..G], g_1 < g_2\} \cup \\ & \{|players_{w_1 g_1} \cap players_{w_2 g_2}| \leq 1 | w_1, w_2 \in [1..W], \\ & \quad w_1 < w_2, g_1, g_2 \in [1..G], g_1 < g_2\} \end{array}$$

where $\wp$ is the powerset. The parametrised graph associated to Golf[W,G,P] is:

$$\begin{array}{ll} V = & \{n_{wgp} | w \in 1..W, g \in 1..G, p \in \wp([1..P*G])\} \\ E_c = & \{n_{wgp} | w \in [1..W], g \in [1..G], |p| \neq P\} \cup \\ & \{\langle n_{wg_1p_1}, n_{wg_2p_2} \rangle | w \in 1..W, g_1, g_2 \in G, g_1 < g_2, \\ & \quad p_1, p_2 \in \wp([1..P*G]), |p_1 \cap p_2| \neq 0)\} \cup \\ & \{\langle n_{w_1g_1p_1}, n_{w_2g_2p_2} \rangle | w_1, w_2 \in 1..W, w_1 < w_2, g_1, g_2 \in G, \\ & \quad g_1 < g_2, p_1, p_2 \in \wp([1..P*G]), |p_1 \cap p_2| > 1\} \\ E_v = & \{\langle n_{wgp,p}, n_{wgp_2} \rangle | w \in 1..W, g \in 1..G, \\ & \quad p_1, p_2 \in \wp([1..P*G]), p_1 \neq p_2\} \end{array}$$

where node n_{wgp} represents literal $players_{wg} = p$. To save space we will not show the concrete form of the generators found for $G[2,2,2]$, but their parametrisation:

A $\{n_{ija} \leftrightarrow n_{ijb} | i \in [1..W], j \in [1..G], a, b \in \wp([1..P*G]), 1 \in a; b = (a \setminus \{1\}) \cup \{2\}\}$
B $\{n_{ija} \leftrightarrow n_{ijb} | i \in [1..W], j \in [1..G], a, b \in \wp([1..P*G]), 2 \in a; b = (a \setminus \{2\}) \cup \{3\}\}$
C $\{n_{ija} \leftrightarrow n_{ijb} | i \in [1..W], j \in [1..G], a, b \in \wp([1..P*G]), 3 \in a; b = (a \setminus \{3\}) \cup \{4\}\}$
D $\{n_{11v} \leftrightarrow n_{12v} | v \in \wp([1..P*G])\}$
E $\{n_{21v} \leftrightarrow n_{22v} | v \in \wp([1..P*G])\}$
F $\{n_{1jv} \leftrightarrow n_{2jv} | j \in [1..G], v \in \wp([1..P*G])\}$

which corresponds to the following symmetries: golfers 1 and 2 can be swapped (**A**), so can golfers 2 and 3 (**B**), golfers 3 and 4 **C**, groups 1 and 2 in week 1 (**D**), groups 1 and 2 in week 2 (**E**), and weeks 1 and 2 (**F**). The generators for $G[3,2,2]$ are simple extensions of those in $G[2,2,2]$, plus two more to cover the additional week: the two groups in week 3 can be swapped (**E1**), and weeks 2 and 3 can be swapped **F1**. Similarly, the generators for $G[4,2,2]$ are simple extensions of those in $G[3,2,2]$ plus, again, two more **E2** and **F2** to cover the additional week. The generators marked as candidates are those in $Old_{\{3,2,2\},\{4,2,2\}}$ are **A,B,C,D,E,F,E1,F1** and **E2**, which leaves **F2** for $New_{\{3,2,2\},\{4,2,2\}}$.

The generators for $G[2,3,2]$ include the simple extension of those in $G[2,2,2]$, plus those needed to include the extra groups and golfers: golfers 4 and 5 can be swapped

(**C1**), so can golfers 5 and 6 (**C2**), groups 2 and 3 in week 1 (**D1**), and groups 2 and 3 in week 2 (**E'1**). Similarly, the generators for $G[2,4,2]$ (once canonicalised by GAP) are simple extensions of those in $G[2,3,2]$ plus, again, another four corresponding to the swapping of golfers 6 and 7 (**C3**), 7 and 8 (**C4**), and groups 3 and 4 in week 1 (**D2**), and 3 and 4 in week 2 (**E'2**). The generators in $Old_{\{2,3,2\},\{2,4,2\}}$ are **A,B,C,D,E,F,C1,C2,C4,D1** and **E'1** are marked as candidates, which leaves **C3, D2** and **E'2** for $New_{\{2,3,2\},\{2,4,2\}}$.

The generators for $G[2,2,3]$ include the simple extension of those in $G[2,2,2]$, plus those needed to include the extra golfers: golfers 4 and 5 can be swapped (**C1**), and so can golfers 5 and 6 (**C2**). The generators for $G[2,2,4]$ are again, those needed to include the extra golfers: golfers 6 and 7 can be swapped (**C3**), and so can golfers 7 and 8 (**C4**), and the simple extension of those in $G[2,2,3]$ with one surprising exception: **C** is replaced by a different and more complex one. This is the case even after canonicalisation by GAP. **C** is,of course, still a symmetry of the group, it is just not returned by Saucy as a generator.

All this has been achieved by our implementations. A more sophisticated parametrisation would also be able to detect the sequences **D:E:E1:E2**, **F:F1:F2**, **D:D:D2**, **E:E'1:E'2**, and **A:B:C:C1:C2:C3:C4**.

6 Conclusions

Exploiting solution symmetries is often essential in finding solutions to CSPs. Currently, the detection of such symmetries is either restricted to problem instances, or incomplete since the existing methods only detect a small class of symmetries and depend on the syntax of the constraints. Our new approach to symmetry detection for CSPs lifts these restrictions: it can detect symmetries in CSP models and it has the power to capture most - if not all - symmetries. Our approach leverages on existing (and future) symmetry detection methods for CSP instances, by generalising their results to models.

Our approach has been implemented and tested on a small set of models, including those discussed in this paper, although some parts require manual intervention. These case studies show that it can detect model symmetries that could previously only be detected for instances. We are currently investigating how to broaden the tool to discover more complex patterns in symmetries of problem instances, to elicit additional candidate model symmetries. We now plan to integrate techniques to validate or reject candidates, such as theorem proving techniques or graph techniques.

References

[Cohen *et al.*, 2005] David Cohen, Peter Jeavons, Christopher Jefferson, Karen E. Petrie, and Barbara M. Smith. Symmetry definitions for constraint satisfaction problems. In Peter van Beek, editor, *LNCS*, volume 3709, pages 17–31, 2005.

[Darga *et al.*, 2004] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry generation for cnf. In *Proc. 41st Design Automation Conf.*, pages 530–534, June 2004.

[de la Banda *et al.*, 2006] Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Proc. CP'06*, 2006.

[Flener *et al.*, 2004] P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. *LOPSTR'03: Revised Selected Papers*, pages 214–232, 2004.

[Frisch *et al.*, 2003] Alan M. Frisch, Ian Miguel, and Toby Walsh. CGRASS: A system for transforming constraint satisfaction problems. In Barry O'Sullivan, editor, *Recent Advances in Constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming*, volume 2627 of *LNCS*, pages 15–30, 2003.

[Frisch *et al.*, 2007] A. M. Frisch, M. Grum, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *IJCAI'07*, 2007.

[GAP, 2006] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.9*, 2006.

[Gent and Smith, 2000] I. P. Gent and B. M. Smith. Symmetry breaking in constraint programming. In *ECAI 2000 14th European Conference on Artificial Intelligence*, 2000.

[Gent *et al.*, 2003] I. P. Gent, W. Harvey, T. Kelsey, and S.Linton. Generic SBDD using computational group theory. In *Proc. CP'03*, pages 333–347, 2003.

[Haselböck, 1993] A. Haselböck. Exploiting interchangeabilities in constraint-satisfaction problems. In *IJCAI'93*, pages 282–289, 1993.

[Hentenryck, 1999] Pascal Van Hentenryck. *The OPL optimization programming language*. MIT Press, Cambridge, MA, USA, 1999.

[Mancini and Cadoli, 2005] T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, 2005.

[Mears *et al.*, 2006] C. Mears, M. Garcia de la Banda, and M. Wallace. On implementing symmetry detection. In *Proc. SymCon 06*, 2006.

[Puget, 2002] Jean-Francois Puget. Symmetry breaking revisited. In Pascal van Hentenryck, editor, *LNCS*, volume 2470, pages 446–461, 2002.

[Puget, 2005] Jean-Francois Puget. Automatic detection of variable and value symmetries. In Peter van Beek, editor, *LNCS*, volume 3709, pages 475–489, 2005.

[Romani and Markov, 2005] Arathi Romani and Igor L. Markov. Automatically exploiting symmetries in constraint programming. *Lecture Notes in Computer Science*, 3419:98–112, 2005.

[Roney-Dougal *et al.*, 2004] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S.Linton. Tractable symmetry breaking using restricted search trees. In *Proc. ECAI'04*, 2004.

[Roy and Pachet, 1998] P. Roy and F. Pachet. Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *ECAI98 Workshop on Non-binary Constraints*, pages 27–33, 1998.

[Sellmann and Hentenryck, 2005] M. Sellmann and P. Van Hentenryck. Structural symmetry breaking. In *Proc. IJCAI'05*, 2005.

[van Hentenryck *et al.*, 2005] P. van Hentenryck, P. Flener, J. Pearson, and M. Ågren. Compositional derivation of symmetries for constraint satisfaction. In *SARA'05*, 2005.

[Walsh, 2006] T. Walsh. General symmetry breaking constraints. In *Proc. CP'06*, 2006.