A Novel Approach For Detecting Symmetries In CSP Models

C. Mears[1], M. Garcia de la Banda[1], M. Wallace[1], B. Demoen[2]

[1] Monash University, Australia
[2] Katholieke Universiteit Leuven, Belgium

Abstract. While several powerful methods exist for automatically detecting symmetries in *instances* of constraint satisfaction problems (CSPs), current methods for detecting symmetries in CSP *models* are limited to the kind of symmetries that can be inferred from the global constraints present in the model. 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 problem instances, and then reasoning on the resulting instance symmetries to infer symmetries of the model. Our results show that this approach deserves further exploration.

1 Introduction

A constraint satisfaction problem (CSP) consists of a set of variables, a set of domains (one per variable), and a set of constraints on the variables. CSPs can often be separated into two parts: the model and the data. The model is a parameterised version of the CSP that, while formally defining the type of variables, domains, and constraints, does not completely determine their number or their values. The data part provides concrete values to the parameters and, as a result, completely determines the number of variables, their domains and the constraints. Thus, while the model represents a *class* of CSPs, the model plus the data specifies an *instance* of that class (i.e., a particular CSP).

For example, the *Latin square problem* of size 3 involves a 3×3 square, where each of the 9 cells in the square takes a value from [1..3], in such a way that each value occurs exactly once in each row and once in each column. The associated CSP can be defined using 9 variables, each with finite domain 1..3, and 18 disequality constraints. Alternatively, it can be separated into a model that is parameterised on the board size N ($N \times N$ variables, each with domain $1..N$, and appropriate constraints), and the data part which simply indicates $N = 3$. Different instances (i.e., CSPs) of the class can be obtained with the same model simply by modifying the value of N in the data.

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, the symmetric search space is known to contain only symmetric solutions (which can be automatically generated without search). If they led to failure, the symmetric search space is known not to contain solutions.

Considerable progress has been made in the automatic detection of symmetries of CSPs and their exploitation in speeding up the search (e.g., [9, 1, 11, 17, 12, 15, 8, 13, 3, 5, 10, 6, 7]. Unfortunately, the most powerful methods ([11, 1]) can only be applied to a CSP, rather than to its model. Therefore, the symmetries detected can only be used to accelerate the solving process for that CSP, and the cost of detecting these symmetries cannot be amortised over all CSPs in the class. Furthermore, the computation cost of these methods grows with the size of the CSP in such a way as to render them impractical for real-size CSPs.

While there are automatic symmetry detection methods for CSP models [14, 16], to our knowledge, they can only detect a relatively small set of "simple" symmetries (i.e., piecewise value and piecewise variable interchangeability), and only from the global constraints in the model. We propose a radically new approach that (1) uses symmetry detection on a series of small CSPs to elicit candidate symmetries, (2) parameterises these candidate symmetries to be defined over the model rather than over a particular CSP, and (3) determines whether these candidates are indeed symmetries of the model – herein referred to as the *parameterised CSP*. Our results show the approach has considerable potential. Furthermore, we believe the approach can be used to infer from the model many other kinds of information useful for optimisation.

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 dom will be omitted.

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.

Example 1. The CSP for the *Latin square problem* of size 3 introduced before can be defined as follows:

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

where variable x_{ij} represents the cell in row i, column j. The set $A = \{x_{21} = 1, x_{22} = 2, x_{23} = 2\}$ is an assignment containing 3 literals. While A satisfies constraint $x_{21} \neq x_{22}$ (since $\{x_{21} = 1, x_{22} = 2\}$ is allowed by it), A does not satisfy $x_{22} \neq x_{23}$ (since $\{x_{22} = 2, x_{23} = 2\}$ is disallowed by it).$\square$

A *solution symmetry* f of a CSP P is a permutation of $lit(P)$ that preserves the set of solutions [1], i.e., a bijection from literals to literals that maps solutions to solutions. Two important kinds of solution symmetries are induced by permuting either variables or values.

A permutation f of the set of variables X induces a permutation p_f of literals by defining $p_f(x = d)$ as $f(x) = d$. A *variable symmetry* is a permutation of the variables whose induced literal permutation is a solution symmetry [10]. Since the inverse of any such permutation is also a symmetry, we will use $\langle x_1, \ldots, x_n\rangle \leftrightarrow \langle x_{1'}, \ldots, x_{n'}\rangle$, where $x_1, \ldots, x_n, x_{1'}, \ldots, x_{n'} \in X$ to denote the symmetry that 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 $x = f_{dom(x)}(v)$. A *value symmetry* is a set of domain permutations whose induced literal permutation is a solution symmetry [10]. We will use $\langle d_{i1}, \ldots, d_{in}\rangle \leftrightarrow \langle d_{i1'}, \ldots, d_{in'}\rangle$, where $\{d_{i1}, \ldots, d_{in}\} = dom(x_i) = \{d_{i1'}, \ldots, d_{in'}\}$, 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 those variable and value symmetries that exist in the CSP.

Several methods [12, 11, 1] have been proposed to automatically detect the symmetries of a CSP by constructing its (hyper-)graph representation, and using graph automorphism techniques on it. Our approach uses the technique of Mears et al. [9] since it is more powerful than that of Puget [11] without being as computationally demanding as that of Cohen et al. [1]. However, any such method can be used. The general 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 2. The Latin square CSP of Example 1 has (a) variable symmetries that swap any 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 $\langle x_{12}, x_{22}, x_{32}\rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33}\rangle$, (b) similar variable symmetries that swap any rows, and (c) variable-value symmetries that transpose the rows, column and value dimensions, and correspond to flipping the 3×3 square using a diagonal. The associated graph (left hand side of Figure 1) has $9{\times}3{=}27$ nodes (labelled $\substack{[i,j]\\k}$) 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.$\square$

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 $\forall\{n_i, \cdots, n_j\} \in E : \{f(n_i), \cdots, f(n_j)\} \in E$. Since, for a given CSP P, the graph has a node for each literal in $lit(P)$, each graph automorphism has a direct interpretation as

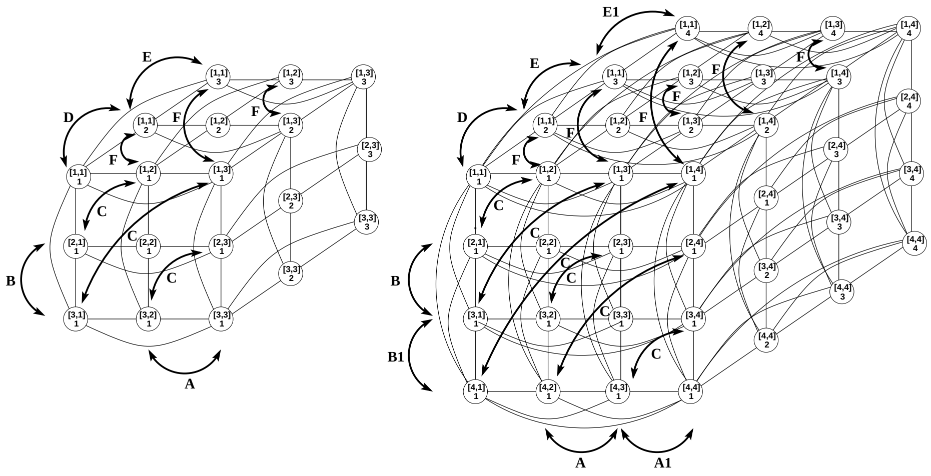

Fig. 1. Graphs and generators for LatinSquare[3] and LatinSquare[4]

a permutation of the literals in *lit*(P) and 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 with a CSP. Standard tools, such as Saucy [2], can compute the automorphisms of a graph and return its symmetry group (i.e., all possible symmetries) by means of a set of generators (a possibly minimal set of symmetries that can be used to generate all others).

Example 3. For the Latin square graph of size 3 given in Example 2, Saucy returns the following set of generators (illustrated in the left hand side of Figure 1):

A $\langle n_{121}, n_{122}, n_{123}, n_{221}, n_{222}, n_{223}, n_{321}, n_{322}, n_{323}\rangle \leftrightarrow$
$\langle n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}, n_{331}, n_{332}, n_{333}\rangle$
B $\langle n_{211}, n_{212}, n_{213}, n_{221}, n_{222}, n_{223}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow$
$\langle n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}, n_{331}, n_{332}, n_{333}\rangle$
C $\langle n_{121}, n_{122}, n_{123}, n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow$
$\langle n_{211}, n_{212}, n_{213}, n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}\rangle$
D $\langle n_{111}, n_{121}, n_{131}, n_{211}, n_{221}, n_{231}, n_{311}, n_{321}, n_{331}\rangle \leftrightarrow$
$\langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{332}\rangle$
E $\langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{333}\rangle \leftrightarrow$
$\langle n_{113}, n_{123}, n_{133}, n_{213}, n_{223}, n_{233}, n_{313}, n_{323}, n_{333}\rangle$
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 node n_{ijk} represents literal $x_{i,j} = k$. **A** states that columns 2 and 3 can be swapped, **B** that rows 2 and 3 can be swapped, **C** that the square can be reflected across the top-left/bottom-right diagonal, **D** that values 1 and 2 can be swapped, **E** that values 2 and 3 can be swapped, and **F** that the second dimension of the square can be swapped with the value dimension. Their combination results in the symmetries given for Example 2 (e.g., to swap columns 1 and 2 ($\langle x_{11}, x_{21}, x_{31}\rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32}\rangle$) apply first **F**, then **D**, and then **F**).$\Box$

3 From CSPs to parameterised CSPs

There is no standard notation to distinguish between a CSP and its parameterised version. Herein, we denote a parameterised CSP as $CSP[Data]$, where $Data$ represents the parameters, and a particular CSP in that class as $CSP[d]$, where d is the value given to $Data$ to yield that CSP. While we will use mathematical notation to specify parameterised CSPs, any high-level modelling language can be used as long as it separates the model from the data, has multi-dimensional arrays of finite domain variables, and supports iteration over them.

Example 4. The parameterised LatinSquare[N] for the CSP of Example 1:

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

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). □

Our aim is to determine the symmetries of every CSP in the class represented by $CSP[Data]$, i.e., the symmetries of $CSP[d]$, for every d possibly given to $Data$. To do so we define the *parameterised* graph $G[Data]$ of $CSP[Data]$ in such as way that, when instantiated by giving a value d to $Data$, $G[d]$ yields the graph of $CSP[d]$. Formally, $G[Data]$ is obtained from $CSP[Data] = (X[Data], D[Data], C[Data], dom[Data])$ as follows:

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

Note that $G[Data]$ is simply a syntactic construct that represents a class of graphs, much as $CSP[Data]$ represents a class of CSPs.

Example 5. The parameterised graph $G[N]$ associated with LatinSquare$[N]$ is as follows. $V[N]$ is defined as $\{n_{ijv} | i, j, v \in [1..N]\}$ where n_{ijv} denotes literal $square_{ij} = v$. $E_v[N]$ is defined as $\{\{n_{ijv_1}, n_{ijv_2}\} | i, j, v_1, v_2 \in [1..N], v_1 \neq v_2\}$, while $E_c[N]$ is obtained by transforming the two constraints in LatinSquare$[N]$ into the set of assignments they disallow:

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

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 appearing in LatinSquare$[N]$. This is important to automate the construction of the edges in $G[N]$ and, as we will see later, to parameterise symmetries of a CSP.$\Box$

We can now give a definition of a parameterised symmetry.

Definition 1. *Given a parameterised $CSP[Data]$ and its parameterised graph $G[Data]$, a parameterised permutation $f[Data]$ is a bijection of the nodes of $G[Data]$. That is, for all values d given to $Data$, $f[d]$ permutes the nodes of $G[d]$. A parameterised symmetry of $CSP[Data]$ is a parameterised permutation $f[Data]$ of the nodes in $G[Data]$ s.t. for all values d given to $Data$, $f[d]$ is a symmetry (i.e., an automorphism) of $G[d]$.*

We denote by $S[Data]$ the group of parameterised symmetries of $CSP[Data]$. Note that for all values d given to $Data$, $S[d]$ is a subset of the symmetries in $CSP[d]$. The subset is proper if some symmetry in $CSP[d]$ does not apply to all other instances of the CSP. In other words, parameterised symmetries must be determined by information explicitly represented in $CSP[Data]$, without requiring information only present in a particular d.

4 A framework for detecting parameterised symmetries

As the main concepts of parameterised CSPs and parameterised symmetries have been introduced, we can now turn to the problem of detecting parameterised symmetries for a class of CSPs. Our approach is based on a generic framework which, given a $CSP[Data]$, performs the following steps:

1. Detect symmetries of $CSP[d]$ for a number of values d given to $Data$,
2. Lift them to obtain parameterised permutations of the literals in $CSP[Data]$,
3. Filter the parameterised permutations to keep only those that are likely to be parameterised symmetries,
4. Prove that the selected parameterised permutations are indeed parameterised symmetries.

Note that while the parameterised CSP is a crucial element of our framework, the parameterised graph is currently used only as a means to define parameterised symmetries. However, as shown later, we plan to use the parameterised graph to obtain a better method than we currently have for step four.

4.1 Step one: Detecting symmetries for some $CSP[d]$

The first step of our generic framework can be realised in different ways by the choice of parameter values and of symmetry detection method. These choices are somewhat mutually dependent. For example, using a powerful symmetry detection method will usually force the parameter values to be small. As mentioned before, our implementation uses the detection method of Mears et al. [9], which returns the group of symmetries in a $CSP[d]$ as a set of group generators[3]. Also, our implementation assumes that the parameter $Data$ is a tuple of k integers, $(p_1, p_2, \ldots, p_k)$ and chooses parameter values d by increasing each component of the tuple individually, starting from some user-defined base tuple (typically the smallest meaningful instance of the class).

Example 6. For LatinSquare[N], $Data$ has a single component: the board size N. If the user provides (3) as the base tuple, we increment the component twice obtaining three values for d: (3), (4), and (5). For the social golfers problem (see Section 5), $Data$ has three components: the number of weeks, groups per week and players per group. If $(2, 2, 2)$ is the base tuple, we increment twice each component to get nine values for d (seven of which are distinct): $(2, 2, 2), (3, 2, 2), (4, 2, 2), (2, 2, 2), (2, 3, 2), (2, 4, 2), (2, 2, 3), (2, 2, 2)$, and $(2, 2, 4)$.□

4.2 Step two: Lifting symmetries to parameterised permutations

This step requires taking every symmetry g detected in step one for any of the $CSP[d]$ considered, and determining one or more parameterised permutation(s) $f[Data]$ for which $f[d] = g$. Since computing $f[Data]$ from g alone is quite a task, our implementation uses a much simpler, although incomplete, method: it first defines a set of "common" parameterised symmetries $Per = \{f_1[Data], \cdots, f_m[Data]\}$, and then checks every generator g against them.

The success of our implementation relies on the parameterised CSPs having literals that can be arranged into an n-dimensional matrix, and having parameterised symmetries that permute particular matrix elements, such as rows or columns. These are the kind of "common" symmetries that we will add to Per.

Consider a $CSP[Data]$ with an n-dimensional matrix-like structure $L[Data]$, whose elements correspond to the literals in $CSP[Data]$ (and, thus to the nodes in $G[Data]$). The exact number of elements in each of the n dimensions of $L[Data]$ depends on the value given to $Data$ and can be obtained by means of a function $Dims[Data] = (d_1, d_2, \ldots, d_n)$, where $d_i, i \in [1..n]$ indicates the exact number of elements in the i^{th} dimension.

Example 7. The parameterised LatinSquare[N] problem has a matrix like structure, since its literals can be arranged into a 3-dimensional matrix where each literal $square_{ij} = k$ is indexed as $L[N]_{i,j,k}$. This is clearly visible in Figure 1, where the only difference between $G[3]$ and $G[4]$ are the exact values of each dimension: $Dims[3] = (3, 3, 3)$ while $Dims[4] = (4, 4, 4)$.□

Parameterised permutations can then be easily expressed as permutations on the elements of $L[Data]$ *without reference to any specific value d given to $Data$*. This allows us to express a parameterised permutation as a single entity, even though each specific instantiation might involve permuting different nodes. Some common parameterised permutations for a $CSP[Data]$ with n-dimensional matrix $L[Data]$ and $Dims[Data] = (d_1, d_2, \ldots, d_n)$ are:

[3] Since the symmetry detection method chosen is incomplete (i.e., might miss some symmetries), our implementation of the generic framework is also incomplete.

- **Value swap:** interchanges values v and v' of the k^{th} dimension (e.g., symmetry represented by generator **D** in Figure 1) and is defined as:
$L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow$
$L[Data]_{i_1,\ldots,i_{k-1},v',i_{k+1},\ldots,i_n}$ $\forall i_j \in [1..d_j], j \in [1..n]$.
- **All values swap:** interchanges all values of the k^{th} dimension (e.g. symmetries represented by generators **D**, **E**, and their combinations in Figure 1) and is defined as: $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow L[Data]_{i_1,\ldots,i_{k-1},v',i_{k+1},\ldots,i_n}$, $\forall v, v' \in [1..d_k], v \neq v'$, $i_j \in [1..d_j], j \in [1..n]$.
- **Dimension invert:** interchanges every value v of the k^{th} dimension with value $n - v + 1$ (e.g., symmetry represented by generator **A** and by generator **A1** in Figure 2) and is defined as: $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow$ $L[Data]_{i_1,\ldots,i_{k-1},n-v+1,i_{k+1},\ldots,i_n}$, $\forall v \in [1..d_k]$, $i_j \in [1..d_j], j \in [1..n]$.
- **Dimension swap:** swaps k^{th} and k'^{th} dimensions (e.g., symmetry represented by generator **B** in Figure 2) and is defined as:
$L[Data]_{i_1,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_{k'-1},i_{k'},i_{k'+1},\ldots,i_n} \leftrightarrow$
$L[Data]_{i_1,\ldots,i_{k-1},i_{k'},i_{k+1},\ldots,i_{k'-1},i_k,i_{k'+1},\ldots,i_n}$, $\forall i_j \in [1..d_j], j \in [1..n]$.

Example 8. The generators found for LatinSquare[3] in Example 3 can be automatically matched to the following parameterised permutations for $L[N]$:

A value swap with $k = 2, v = 2, v' = 3$: $L[N]_{i2l} \leftrightarrow L[N]_{i3l}, \forall i, l \in [1..N]$
B value swap with $k = 1, v = 2, v' = 3$: $L[N]_{2jl} \leftrightarrow L[N]_{3jl}, \forall j, l \in [1..N]$
C dimension swap with $k = 1, k' = 2$: $L[N]_{ijl} \leftrightarrow L[N]_{jil}, \forall i, j, l \in [1..N]$
D value swap with $k = 3, v = 1, v' = 2$: $L[N]_{ij1} \leftrightarrow L[N]_{ij2}, \forall i, j \in [1..N]$
E value swap with $k = 3, v = 2, v' = 3$: $L[N]_{ij2} \leftrightarrow L[N]_{ij3}, \forall i, j \in [1..N]$
F dimension swap with $k = 2, k' = 3$: $L[N]_{ijl} \leftrightarrow L[N]_{ikl}, \forall i, j, l \in [1..N]$

Consider the graph $G[4]$ associated with 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 those found for $G[3]$. For example, the extension of **A** is:

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:

A1 $\langle n_{131}, n_{132}, n_{133}, n_{134}, n_{231}, n_{232}, \ldots, n_{331}, \ldots, n_{431}, \ldots \rangle \leftrightarrow$
$\langle n_{141}, n_{142}, n_{143}, n_{144}, n_{241}, n_{242}, \ldots, n_{341}, \ldots, n_{441}, \ldots \rangle$
B1 $\langle n_{311}, n_{312}, n_{313}, n_{314}, n_{321}, n_{322}, \ldots, n_{331}, \ldots, n_{341}, \ldots \rangle \leftrightarrow$
$\langle n_{411}, n_{412}, n_{413}, n_{414}, n_{421}, n_{422}, \ldots, n_{431}, \ldots, n_{441}, \ldots \rangle$
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 **A, B, C, D, E** and **F** in $G[4]$ match the parameterised permutations used for $G[3]$, while **A1, B1** and **E1** match value swap with:

A1 $k = 2, v = 3, v' = 4$: $L[N]_{i3l} \leftrightarrow L[N]_{i4l}, \forall i, l \in [1..N]$
B1 $k = 1, v = 3, v' = 4$: $L[N]_{3jl} \leftrightarrow L[N]_{4jl}, \forall j, l \in [1..N]$
E1 $k = 3, v = 3, v' = 4$: $L[N]_{ij3} \leftrightarrow L[N]_{ij4}, \forall i, j \in [1..N]$

The generators found by Saucy for LatinSquare[5] are the simple extensions of **A, A1, B, B1, C, D, E, E1** and **F** (which can be parameterised as before), plus three more **A2, B2**, and **E2**, which can be parameterised as:

A2 $k = 2, v = 4, v' = 5$: $L[N]_{i3l} \leftrightarrow L[N]_{i4l}, \forall i, l \in [1..N]$
B2 $k = 1, v = 4, v' = 5$: $L[N]_{3jl} \leftrightarrow L[N]_{4jl}, \forall j, l \in [1..N]$
E2 $k = 3, v = 4, v' = 5$: $L[N]_{ij3} \leftrightarrow L[N]_{ij4}, \forall i, j \in [1..N]$□

Considering a symmetry g in isolation is not always productive. This is because some parameterised permutation patterns, when instantiated, correspond to a group of symmetries rather than to a single symmetry. For example, the "all values swap" pattern (which interchanges all values in a dimension) is a combination of at least two generator symmetries. Thus, to detect such a parameterised pattern we cannot simply parameterise each symmetry on its own; we must consider groups of symmetries $\{g_1, \cdots, g_m\}$ such that $f[d] = \{g_1, \cdots, g_m\}$.

For the "all value swap" case, we group symmetries by keeping track of any pair of value-swap pattern symmetries which operate on the same dimension and whose interchanged values overlap. These are combined into a single symmetry stating that all values involved can be freely interchanged. Our implementation considers the "all values swap" pattern matched if, by applying this kind of combination until a fixpoint is reached, we obtain a symmetry that interchanges all $[1..d_k]$, where d_k is the value returned by $Dims[d]$ for dimension k.

Example 9. The generators **D** and **E** for LatinSquare[3] form an instance of the "all value swap" pattern $L[N]_{ijv} \leftrightarrow L[N]_{ijv'}, \forall v, v' \in [1..N], v \neq v', i, j \in [1..N]$. The generators **D**, **E** and **E1** for LatinSquare[4] form the same pattern.□

4.3 Step three: Filtering parameterised permutations

Step two identifies our candidate parameterised symmetries. However, it is likely that some of these candidates apply only to a few instances, rather than to the entire class. We would like to eliminate unlikely permutations before performing the (possibly expensive) proof step. Our implementation uses a simple (and again incomplete) heuristic which selects as likely candidates the intersection of the parameterised permutations present in all tested instances.

Unfortunately, the success of such an intersection relies on Saucy returning the same (or equivalent) set of generators for each $CSP[d]$. This is because, as mentioned before, our implementation only attempts to parameterise the generators returned by Saucy (as opposed to every symmetry in the group), and a group can be obtained from many different sets of generators. We can solve this problem as follows. If a particular parameterised permutation is found in more than one instance but not in all, we check the group of symmetries of the other instances to see if the permutation is, in fact, present. This is done via the GAP system for computational group theory [4]. If the parameterised permutation is indeed found in all instances, it is marked as a candidate.

Example 10. Consider the social golfers problem with values of d being $(2, 2, 2)$, $(3, 2, 2)$, $(4, 2, 2)$, $(2, 3, 2)$, $(2, 4, 2)$, $(2, 2, 3)$, $(2, 2, 4)$. Our implementation finds an instance of the "all value swap" pattern for the third dimension (golfers are interchangeable) for every value of d. However, the "all value swap" pattern for the first dimension (the weeks are interchangeable) is found for only 5 out

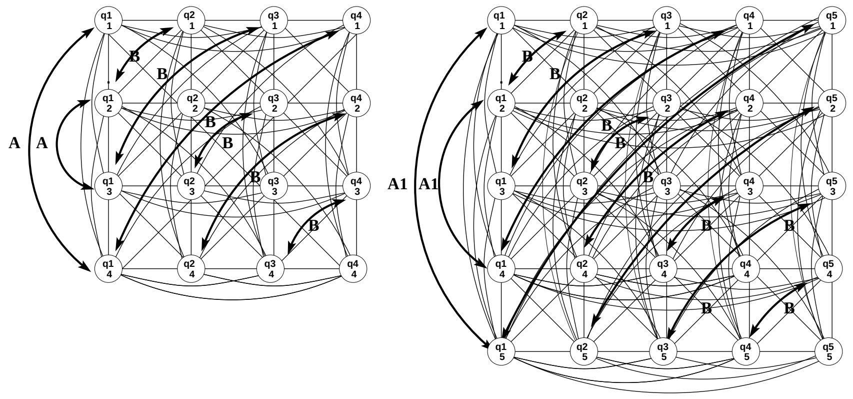

Fig. 2. Graph instances for Queens[4] and Queens[5]

of the 7 values of d, due to the particular generators given by Saucy. Searching explicitly for this pattern in the groups found for the other values of d shows that it is indeed present in all of them and can thus be considered a likely candidate.

4.4 Step four: Proving class symmetries

This last step can be achieved, for example, by first representing both the parameterised CSP and the candidate parameterised permutation in the logic formalism described in [8], and then making use of theorem proving techniques. Of course, such a technique is in general undecidable. We are currently exploring an alternative approach that we hope will be more successful: to use graph techniques to prove that our likely candidate is an automorphism of the parameterised graph $G[Data]$. This is however not straightforward, since $G[Data]$ is not really a graph, but a syntactic construct that represents a class of graphs.

5 Detailed Examples

We have seen how our current implementation automatically detects as likely candidates all parameterised symmetries in LatinSquare[N]. This is, however, not always the case. Here we provide three other examples: Queens, for which it again detects all parameterised symmetries as likely candidates, Social golfers, for which it also detects all symmetries (after adding a new pattern), and Golomb ruler, for which it fails to detect any likely candidate.
Queens: aims at positioning N queens on an $N \times N$ chess board without attacking each other. The following parameterised CSP Queens[N] uses N integer variables (each being the row in which the queen appears) 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 parameterised graph $G[N] = (V[N], E_c[N] \cup E_v[N])$ is:

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

where node q_{iv} represents literal $q_i = v$. Given the initial base tuple (4), our implementation generates $G[4]$, $G[5]$ and $G[6]$. Figure 2 shows the graph instances $G[4]$ and $G[5]$, together with the generators found by Saucy. For $G[4]$ it finds:

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

which can be parameterised to match:

A dimension invert with $k = 2$: $L[N]_{iv} \leftrightarrow L[N]_{i(N-v+1)}, \forall v, i \in [1..N]$
B dimension swap with $k = 1$ and $k' = 2$: $L[N]_{ij} \leftrightarrow L[N]_{ji}, \forall i, j \in [1..N]$

and for $G[5]$ Saucy finds:

A1 $\langle q_{11}, q_{12}, q_{21}, q_{22}, q_{31}, q_{32}, q_{41}, q_{42}, q_{51}, q_{52} \rangle \leftrightarrow$
$\langle q_{15}, q_{14}, q_{25}, q_{24}, q_{35}, q_{34}, q_{45}, q_{44}, q_{55}, q_{54} \rangle$
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 matches the same dimension swap pattern), and **A1** is a new generator that matches the same dimension invert pattern as **A**. The generators found for $G[6]$ are, again, an extension of **B** that matches the dimension swap pattern, and a new generator **A2** that matches the same pattern as **A** and **A1**. The intersection of the patterns results in both being marked as likely candidates.

Social Golfers: aims at building a schedule of W weeks, with G equally-sized groups per week, and P golfers per group, such that each pair of golfers may play in the same group at most once. A parameterised CSP Golf$[W, G, P]$ is:

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

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

$$\begin{aligned} V[W,G,P] = {} & \{n_{wgp} | w \in [1..W], g \in [1..G], p \in \wp([1..P * G])\} \\ E_c[W,G,P] = {} & \{n_{wgp} | w \in [1..W], g \in [1..G], |p| \neq P\} \cup \\ & \{\langle n_{wg_1p_1}, n_{wg_2p_2} \rangle | w \in [1..W], g_1, g_2 \in G, g_1 < g_2, \end{aligned}$$

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

where node n_{wgp} represents literal $\mathit{players}_{wg} = p$. The parameterised versions of the generators found for $G[2,2,2]$ are:

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 $\{n_{1jv} \leftrightarrow n_{2jv} | j \in [1..G], v \in \wp([1..P*G])\}$

Generators **A**, **B** and **C** represent symmetries that swap golfers 1 with 2, 2 with 3, and 3 with 4, respectively. Taken together, our implementation detects the combined all value swap permutation pattern that states that all golfers are interchangeable. Generator **F** represents the symmetry that swaps week 1 and week 2. This trivially matches the all value swap pattern that states that all weeks are interchangeable, and also the dimension invert pattern that reflects the weeks. Generators **D** and **E** represent symmetries that swap groups 1 and 2 within week 1, and within week 2, respectively. Our implementation did not consider parameterised patterns that perform a swap on only a subset of the literals and, thus, failed to detect such pattern as likely candidate. However, once we extended the set of patterns to include one that represents the interchanging of values within a particular row or column, this symmetry was captured.

The generators for $G[2,3,2]$, $G[2,2,3]$, $G[2,4,2]$ and $G[2,2,4]$ include the extended versions of generators **A** to **F** in $G[2,2,2]$, plus additional generators representing the interchangeability of the extra golfers, and of the extra groups. As before, our implementation detects the combined all value swap pattern that states that all golfers are interchangeable and that all weeks are interchangeable. With the inclusion of the pattern mentioned above, the interchangeability of groups within each week is also marked as likely candidate.

The generators for $G[3,2,2]$ include the extended versions of **A**, **B**, **C**, **D** and **E**. However, Saucy produces generators that do not have a simple parameterisation. The situation for $G[4,2,2]$ is similar. But since the weeks were found to be interchangeable in all of the other instances, the implementation consults GAP to check whether this holds for $[3,2,2]$ and $[4,2,2]$, even though the generators from Saucy don't directly correspond to it. GAP indicates that it does and, therefore, the symmetry is marked as a likely candidate.

Golomb ruler: is defined as a set of N integers (marks on the ruler) $a_1, \ldots, a_N$ 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 parameterised CSP 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 $[1..N^2]$.

$$\begin{aligned}
X[N] = &\quad \{mark_i | i \in [0..N]\} \cup \{\mathit{diff}_{ij} | i \in [1..N], j \in [i+1..N]\}\\
D[N] = &\quad \{[0..N^2], [1..N^2]\}\\
C[N] = &\quad \{mark_i - mark_j = \mathit{diff}_{ij} | i \in [1..N], j \in [i+1..N]\} \cup\\
&\quad \{\mathit{diff}_{ij} \neq \mathit{diff}_{ik} | i, j \in [1..N], k \in [j+1..N]\}\\
dom(m_i) &= [0..N^2] \;;\; dom(d_{ij}) = [1..N^2]
\end{aligned}$$

The parameterised graph associated with Golomb$[N]$ is:

$$\begin{aligned}
V[N] = &\{m_{iv} | i \in [1..N], v \in [0..N^2]\} \cup\\
&\{d_{jiv} | i \in [1..N], j \in [(i+1)..N], v \in [1..N^2]\}\\
E_c[N] = &\{\{m_{iv_1}, m_{jv_2}, d_{ijv_3}\} | i \in [1..N], j \in [(i+1)..N],\ v_1, v_2, v_3 \in [1..N^2], v_1 - v_2 \neq v_3\} \cup\\
&\{\{d_{ijv}, d_{ijv}\} | i \in [1..N], j \in [(i+1)..N], v \in [1..N^2]\}\\
E_v[N] = &\{(m_{iv_1}, m_{iv_2}) | i \in [1..N], v_1, v_2 \in [1..N^2], v_1 \neq v_2\} \cup\\
&\{(d_{ijv_1}, d_{ijv_2}) | i \in [1..N], j \in [(i+1)..N], v_1, v_2 \in [1..N^2], v_1 \neq v_2\}
\end{aligned}$$

where node m_{iv} represents literal $mark_i = v$ and node d_{ijv} literal $\mathit{diffs}_{ij} = v$. The generator found by Saucy 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. This symmetry involves variables from two separate matrices, d_{ij} and m_i, and our simple implementation cannot yet handle this. Even if we only consider the search variables m_i, our implementation would need to obtain for $G[3]$, $G[4]$ and $G[5]$ the pattern $\{m_{iv} \leftrightarrow m_{jv'} | i, j \in [1..N], i = N - j + 1, v, v' \in [0..N^2], v = N^2 - v' + 1\}$. Since our implementation currently does not take this pattern into account, it cannot recognise the symmetry as likely candidate.

6 Results

Let us evaluate our simple implementation (which includes the patterns described in Section 4.2 plus the additional pattern described for Social Golfers) over a set of problems that include those discussed earlier, plus the following.
Balanced Incomplete Block Design: with parameters (v, b, k, r, λ), where the task is to arrange v objects into b blocks such that each block has exactly k objects, each object is in exactly r blocks, and every pair of objects occurs together in λ blocks. The objects are interchangeable and the blocks are interchangeable.
Graceful Graph: with parameters (m, n), where the edges (a, b) of the graph $K_m \times P_n$ are labeled by $|a - b|$, and there is no two edges with the same label. The corresponding vertices in each clique are simultaneously interchangeable, the order of the cliques is reversible, and the values are reversible.
N × N queens: where an $N \times N$ chessboard is coloured with N colours, so that a pair of queens in any two squares of the same colour do not attack each other. The symmetries are those of the chessboard, plus the colours are interchangeable.
Queens (bool): which uses a Boolean matrix model for the Queens problem of Section 5. The symmetries are those of the chessboard.

Table 1. Symmetry detection results

Problem	Tuple	Amount	Symmetries	Time	Instance
BIBD	(2,2,2,2,2)	+3	objects ✓ blocks ✓	19.0	20%
Social Golfers	(2,2,2)	+2	rows ✓ groups ✓ players ✓	376.4	96%
Golomb Ruler	(3)	+3	flip X	6.7	99%
Graceful Graph	(2,2)	+3	intra-clique ✓ path-reverse ✓ value ✓	9.0	44%
Latin Square	(3)	+3	dimensions ✓ value ✓	13.7	10%
N $\times$ N queens	(4)	+3	chessboard ✓ colours ✓	8.0	21%
Queens (int)	(8)	+3	chessboard ✓	3.6	36%
Queens (bool)	(8)	+3	chessboard ✓	5.4	64%
Steiner Triples	(3)	+3	triples ✓ value ✓	16.8	32%

Steiner Triples: where the task is to find $\frac{n(n-1)}{6}$ triples of distinct integers from 1 to n, such that any pair of triples has at most one element in common. The triples are interchangeable and the values are interchangeable.

Table 1 shows the results, where the columns indicate the problem name, the base tuple, the amount by which each component is increased, the known symmetries and whether they are found by our implementation, the total running time in seconds, and the percentage of that time spent in detection (as opposed to parametrisation). The experiments were run on an dual Intel Core 2 1.86GHz computer with 1GB of memory. No effort has been made to optimise detection time; the times are included simply to show the practicality of the approach.

7 Conclusions

The automatic detection of CSP symmetries is currently either restricted to problem instances, or limited to the class of symmetries that can be inferred from the global constraints present in the model. This paper provides a radically new framework that takes advantage of existing (and future) powerful detection methods defined for problem instances, by generalising their results to models without requiring them to use any particular syntax. We provide a very simple (and incomplete) implementation that requires the problem to have matrix-like structure and only considers a pre-determined number of model symmetries (those that correspond to permutations of the objects in the matrix). While this is a very limited implementation of the general framework, it is nonetheless capable of detecting symmetries that could previously only be detected for instances. Of course, more complete implementations of the framework will be able to detect even more kinds of symmetries.

We now plan to integrate in our implementation techniques to validate or reject likely candidates. While theorem proving techniques are an obvious possibility, we are also investigating graph techniques that rely on the parameterised graph and which we think will be more efficient and complete.

References

1. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K.E., Smith, B.M.: Symmetry Definitions for Constraint Satisfaction Problems. In: Van Beek, P. (ed.), CP 2005. LNCS, vol. 3709, pp. 17–31. Springer, Heidelberg (2005)
2. Darga, P.T., Liffiton, M.H., Sakallah, K.A., Markov, I.L.: Exploiting Structure in Symmetry Generation for CNF. In: 41st Design Automation Conference, pp. 530–534, 2004
3. Frisch, A.M., Miguel, I., and Walsh, T.: CGRASS: A System for Transforming Constraint Satisfaction Problems. In: O'Sullivan, B. (ed.), International Workshop on Constraint Solving and Constraint Logic Programming. LNCS, vol. 2627, pp. 15–30. Springer, 2003.
4. The GAP Group. GAP – Groups, Algorithms, and Programming, Version 4.4.9. 2006.
5. Gent, I.P., Harvey, W., Kelsey, T., Linton, S.: Generic SBDD using Computational Group Theory. In CP 2003, pp. 333–347. Springer, 2003.
6. Gent, I.P., Smith, B.M.: Symmetry Breaking in Constraint Programming. In ECAI 2000 14th European Conference on Artificial Intelligence, 2000.
7. Haselböck, A.: Exploiting Interchangeabilities in Constraint-Satisfaction Problems. In IJCAI'93, pp. 282–289. 1993.
8. Mancini, T., Cadoli, M.: Detecting and Breaking Symmetries by Reasoning on Problem Specifications. In International Symposium on Abstraction, Reformulation and Approximation (SARA 2005). 2005.
9. Mears, C., Garcia de la Banda, M., Wallace, M.: On Implementing Symmetry Detection. In SymCon06. 2006.
10. Puget, J-F.: Symmetry Breaking Revisited. In Van Hentenryck, P. (ed.), CP 2002. LNCS, vol. 2470, pp. 446–461. Springer, 2002.
11. Puget, J-F.: Automatic Detection of Variable and Value Symmetries. In Van Beek, P. (ed.), CP 2005. LNCS, vol. 3709, pp. 475–489. Springer, 2005.
12. Romani, A., Markov, I.L.: Automatically Exploiting Symmetries in Constraint Programming. In LNCS vol. 3149, pp. 98–112. Springer, 2005.
13. Roney-Dougal, C.M., Gent, I.P., Kelsey, T., Linton, S.: Tractable Symmetry Breaking using Restricted Search Trees. In ECAI'04. 2004.
14. Roy, P., Pachet, F.: Using Symmetry of Global Constraints to Speed Up the Resolution of Constraint Satisfaction Problems. In ECAI98 Workshop on Non-binary Constraints, pp. 27–33. 1998.
15. Sellmann, M., Van Hentenryck, P.: Structural Symmetry Breaking. In IJCAI'05. 2005.
16. Van Hentenryck, P., Flener, P., Pearson, J., Ågren, M.: Compositional Derivation of Symmetries for Constraint Satisfaction. In SARA'05. 2005.
17. Walsh, T.: General Symmetry Breaking Constraints. In Benhamou, F. (ed.) CP 2006. LNCS, vol. 4204. Springer, 2006.