Constraints (2009) 14:443–477
DOI 10.1007/s10601-008-9057-9

On implementing symmetry detection

C. Mears · M. Garcia de la Banda · M. Wallace

Published online: 23 July 2008
© Springer Science + Business Media, LLC 2008

Abstract Automatic symmetry detection has received a significant amount of interest, which has resulted in a large number of proposed methods. This paper reports on our experiences while implementing the approach of Puget (CP2005, LNCS, vol. 3709, pp. 475–489. Springer, 2005). In particular, it proposes a modification to the approach to deal with general expressions, discusses the insights gained, and gives the results of an experimental evaluation of the accuracy and efficiency of the approach.

Keywords Automatic symmetry detection · Graph automorphism

1 Introduction

Symmetries in constraint satisfaction problems (CSPs) can be used to speed up the search for solutions. This is achieved by avoiding the exploration of areas of the search space that are symmetric to areas that are already (or will be) explored [3]. This is correct because if the explored area led to failure, the symmetric area must also lead to failure. If, on the other hand, it led to a solution s, the symmetric area is known to only contain solutions that can be more efficiently generated by applying the symmetries to s.

A common classification of symmetries distinguishes *variable* symmetries, *value* symmetries, and *variable-value* symmetries, which refer respectively to permutations

C. Mears (✉) · M. Garcia de la Banda · M. Wallace
Clayton School of IT, Monash University, Melbourne, Australia
e-mail: cmears@mail.csse.monash.edu.au

M. Garcia de la Banda
e-mail: mbanda@mail.csse.monash.edu.au

M. Wallace
e-mail: wallace@mail.csse.monash.edu.au

among only the variables in the CSP, among only the values of each variable, and among variable-value pairs, which preserve the set of solutions [12]. Related to these concepts are the notions of value *interchangeability* [5] where all values of a variable are equivalent (i.e., every permutation among the values is a symmetry), *piecewise* value interchangeability [11] where only some subsets of values are interchangeable, and their direct variable counterparts (variable interchangeability and piecewise variable interchangeability).

The importance of automatically detecting symmetries has generated a considerable amount of interest and has resulted in a number of different methods that automatically detect one or more kinds of symmetries. These methods can be split into two main categories: those that detect the symmetries of a given CSP (or CSP *instance*) [3, 5, 6, 13, 15], and those that detect symmetries of a *class* of CSPs (or CSP *model*) [11, 16]. Each category has advantages and disadvantages. The main advantage of model-based methods is that the symmetry detection process only needs to be performed once for the whole class, since each symmetry detected for a CSP model is known to also be a symmetry of all its CSP instances. Instead, instance-based methods need to be re-run for each instance. This creates efficiency problems, especially since the more powerful methods (such as [13]) can become impractical for large instances. The main advantage of instance-based methods is that they can take the particular instance data into account and, therefore, might be able to detect a greater number of symmetries (including some not present in the model). Furthermore, current model-based methods are relatively limited in the kind of symmetries they detect (piecewise variable and value interchangeability) and in the accuracy of their detection process (which depends on the problem being modelled using global constraints, and on the accuracy of their composition functions).

Our long term aim is to develop a method that (a) detects a broader range of symmetries than those detected by current model-based methods, (b) detects symmetries that apply to the CSP model, (c) does not require the model to use a particular problem formulation (such as global constraints), and (d) is practical. In order to achieve this we decided to build on powerful instance-based symmetry detection techniques to discover symmetries for models. The idea is to (1) use powerful instance-based symmetry detection methods on a series of small problem instances to elicit candidate symmetries, (2) parametrise these candidate symmetries to be defined over the model rather than over the data of any particular instance, and (3) determine whether these are indeed symmetries of the model.

This paper presents a first step towards this long term aim: the implementation of a powerful instance-based symmetry detection method. In particular, it reports on work performed while studying three different graph representations of CSP instances: Puget's extensional method introduced in [13], and the microstructure and microstructure complement introduced by Jégou [9] and extended by Cohen et al. [2]. The microstructure complement is a very powerful and general graph representation capable of detecting symmetries of a CSP by representing the set of disallowed assignments of each constraint in the CSP. Cohen et al. prove that automorphisms of this graph correspond to symmetries of the CSP. However, the requirement to only use disallowed assignments means the graph can be unnecessarily big. Instead, the microstructure requires each constraint to be represented by the set of allowed assignments. However, automorphisms of this graph have not been proved to correspond to symmetries of the CSP. In fact, we later show this not to be the case.

Puget's extensional approach admits a more flexible representation of constraints in which one can choose to use the allowed or the disallowed assignments of each constraint in the CSP. The disadvantage of this approach is that some of its features, such as the explicit representation of variable nodes and constraint nodes, limit the kind of symmetries that can be detected.

Our aim is to develop a graph representation that combines the best of these two approaches. In particular, the contributions of this paper are as follows. We first introduce the *allowed assignments* graph, which is extends the microstructure in such a way that its automorphisms can be proved to correspond to symmetries of the CSP while, at the same time, avoiding too much growth in the size of the graph. We then define a second graph representation, the *full assignments* graph, which (as Puget's extensional method) can use the set of allowed or disallowed assignments depending on the particular constraint considered, but without restricting the symmetries that can be represented by the graph. We also prove that every automorphism of the full assignment graph corresponds to a symmetry of the CSP, while there may exist symmetries which are not automorphisms of the graph.

We then propose two techniques for pruning different kinds of nodes in the full assignment graph (that can also be applied to the other two): one based on establishing *n*-ary arc-consistency, and another based on reducing the arity of global constraints by using a logically equivalent conjunction of constraints of smaller arity. We prove that the former approach does not eliminate any variable or value symmetries (although it might eliminate non-compositional variable-value symmetries) if all constraints represented in the graph have different scope. We also show how the latter technique can lead to an increase in the number of symmetries detected.

When compared to Puget's extensional representation [13], the full assignments graph increases the number of constraint symmetries that can be detected by, for example, dropping the use of variable nodes and eliminating representational differences among constraints. Furthermore, it is less dependant on constraint syntax. The comparison with Puget's Boolean representation, also given in [13], is less clear and will be discussed in detail in Section 2.3. Finally, the paper gives the results of an experimental evaluation that compares Puget's extensional and Boolean representations with ours over a number of benchmarks.

The rest of the paper proceeds as follows. In the next section we discuss previous work on CSP symmetry detection and symmetry breaking, including a detailed discussion of Puget's method for constructing the graph associated with a given CSP. Section 3 provides the definition of the allowed assignments graph, the disallowed assignments graph, and the full assignments graph, together with our insights into their properties and the relationship with related work. Section 4 describes our two techniques to represent graphs of CSPs more concisely, and the effects these have on the detected symmetries. Section 5 presents the results of our experimental evaluation. Finally, Section 6 presents our conclusions.

2 Background

This section introduces the terminology to be used in the paper and provides a summary of Puget's method [13] to automatically detect symmetries.

2.1 CSP symmetry

A CSP is a triple (X, D, C) where X represents a set of variables, D a set of domains, C a set of constraints, and where each variable $x_i \in X$ is associated with a finite domain $D_i \in D$ of potential values. By an abuse of notation, if $\forall D_i, D_j \in D : D_i = D_j$, we will then make D equal to D_i, i.e., we will present the CSP in the form (X, D_i, C).

A *literal* is of the form $x_i = d_i$ where $x_i \in X$ and $d_i \in D_i$. For any literal l of the form $x_i = d_i$, we will use $var(l)$ to denote its variable x_i. An *assignment* A is a set of literals. An assignment *over a set of variables* $V \subseteq X$ has exactly one literal $x_i = d_i$ for each variable $x_i \in V$. An assignment over X is called a *complete* assignment.

A constraint c is defined over a set of variables which is called its *scope*, and is written $vars(c)$. A constraint c 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 which satisfies every constraint in C.

A constraint $c \in C$ can be represented *extensionally* by the set of allowed assignments over $vars(c)$, or *intensionally* by a function that, given an assignment A, returns `true` if A satisfies c, and `false` otherwise. Since we only deal with finite domains and global constraints whose arguments are known, any intensional constraint can be converted into its extensional equivalent.

A *solution symmetry* f is a permutation of literals that preserves the set of solutions [2]. In other words, it is a bijection from literals to literals that maps solutions to solutions. Therefore, if $\{l_1, \ldots, l_n\}$ is a solution, then $\{f(l_1), \ldots f(l_n)\}$ is also a solution, and if A_1 and A_2 are two distinct complete assignments, then $f(A_1)$ and $f(A_2)$ are also distinct. Consequently, we can also say that for any solution symmetry f, assignment A is a solution if and only if $f(A)$ is. A *constraint symmetry* is a solution symmetry that preserves the constraints of the CSP.

We now introduce some common, orthogonal classes of symmetries. A *variable symmetry* is a permutation of the variables that preserves the constraints or the solutions [12]. 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 $\{x_1, \ldots, x_n\} = X = \{x_{1'}, \ldots, x_{n'}\}$, to denote the variable symmetry which maps every x_i to $x_{i'}$ (or every $x_{i'}$ to x_i). For simplicity, if we have $\{x_1, \ldots, x_k\}, \{x_{1'}, \ldots, x_{k'}\} \subset X$, then $\langle x_1, \ldots, x_k \rangle \leftrightarrow \langle x_{1'}, \ldots, x_{k'} \rangle$ denotes the symmetry which maps each x_i to $x_{i'}$ leaving the remaining variables unchanged.

A *value symmetry* is a permutation within the sets in D (i.e., a bijection from the values of a variable to values of that variable) that preserves the constraints or the solutions [12]. We will use $\langle d_{i1}, d_{i2}, \ldots, d_{in} \rangle \leftrightarrow \langle d_{i1'}, d_{i2'}, \ldots, d_{in'} \rangle$, where $\{d_{i1}, d_{i2}, \ldots, d_{in}\} = D_i = \{d_{i1'}, d_{i2'}, \ldots, d_{in'}\}$, to denote a value symmetry for a given variable $x_i \in X$. A *variable-value* symmetry is a permutation of the literals (i.e. the set $V \times D$) that preserves the constraints or the solutions. Note that a variable-value symmetry of a CSP is not necessarily a composition of a variable symmetry and a value symmetry of that CSP (i.e., one or both might not be symmetries of that CSP). These kind of symmetries will be referred to as *non-compositional* variable-value symmetries.

Example 1 The common N-queens problem requires the placement of N queens on an $N \times N$ chessboard such that no queen attacks another. We can model this problem using one integer variable x_i per row i in the board so that each value, $d \in \{1, \ldots, N\}$, represents the column position of x_i in row i.

The corresponding CSP is $(\{x_1, \ldots, x_N\}, \{1, \ldots, N\}, C)$, where $\forall i, j$ s.t. $i < j$ we have $\{x_i \neq x_j, |x_i - x_j| \neq j - i\} \subseteq C$. This CSP has the variable symmetry $\langle x_1, \ldots, x_N \rangle \leftrightarrow \langle x_N, \ldots, x_1 \rangle$ (representing the reflection around a horizontal axis through the centre of the board), the value symmetry $\langle 1, \ldots, N \rangle \leftrightarrow \langle N, \ldots, 1 \rangle$ (vertical axis), and it also has the variable-value symmetry that maps every $x_i = j$ to $x_j = i$ (top-left/bottom-right diagonal). Note that the last is a non-compositional variable-value symmetry that does not result from composing the previous two.

Given a graph represented by the tuple $\langle V, E \rangle$, where V is a set of nodes, and E a set of unweighted and undirected edges, an *automorphism* f of graph $\langle V, E \rangle$ is a permutation of the nodes such that $\forall (n_i, n_j) \in E : (f(n_i), f(n_j)) \in E$.

2.2 Puget's coloured graphs

The method presented by Puget in [13] has two steps. The first takes a CSP and constructs a *coloured* graph, i.e., a graph represented by the triple $\langle V, E, c \rangle$ where V and E are as before, and c is a map from V to colours. The second step finds the automorphisms of this graph *that also preserve the colours*, i.e., that only interchange nodes of the same colour. Formally, an automorphism f of graph $\langle V, E, c \rangle$ is a permutation of the nodes such that $\forall (n_i, n_j) \in E : (f(n_i), f(n_j)) \in E$ and $\forall n \in V, c(f(n)) = c(n)$.

Example 2 The graph shown in Fig. 1a can be reflected across its vertical axis resulting in that of Fig. 1b, where the dashed arrows indicate the node permutation used for this reflection. Since the graph edges are preserved, the permutation is an automorphism. Consider now the graph shown in Fig. 1c where colours are represented by shading patterns. Its reflection over the horizontal axis results in

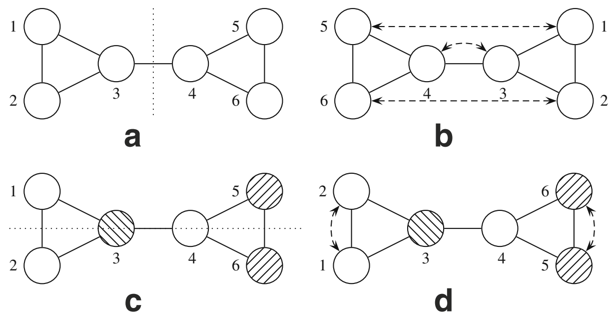


Fig. 1 Two graphs and one of their possible automorphisms

Fig. 1d, where the dashed arrows again indicate the associated node permutation. This permutation is also an automorphism. Note that reflecting the graph across the vertical axis no longer results in an automorphism due to the node colours.

In Puget's method, the coloured graph associated with the CSP (X, D, C) can be constructed as follows. First, a *variable* node is created for each variable $x_i \in X$, and a *constraint* node is created for each constraint $c \in C$. These nodes are coloured as follows: all variable nodes have the same unique colour, and all constraint nodes representing a particular "kind" of constraint have the same (also unique) colour. Constraints can then be represented using two different methods based on an *intensional* or an *extensional* representation, respectively. For the intensional constraint representation, an edge is added between each constraint node and the variable nodes in its scope. Dummy nodes might be required to break symmetries that do not occur in the constraint. Automorphisms of this graph correspond to variable symmetries.

For the extensional constraint representation, a *value* node is introduced for each value of each variable in the scope of the constraint, and an *assignment* node is created for each allowed assignment of each constraint. Edges connect each value node to its variable node, each assignment node to the value node representing each variable-value literal occurring in the assignment, and each assignment node to its constraint node. These nodes are coloured as follows: all value nodes for a variable must have the same unique colour if we want to detect value symmetries, while all value nodes (regardless of variable) must have the same unique colour if we also want to detect variable symmetries. All assignment nodes have the same unique colour. As indicated by Puget, if the set of allowed assignments contains many more elements than the set of disallowed assignments, then the latter can be used to construct a smaller graph. Automorphisms of this graph correspond to variable symmetries, value symmetries and compositional variable-value symmetries.

Example 3 Consider the CSP $(\{x, y\}, \{1, 2, 3, 4\}, \{x \geq y\})$. Figure 2a, b show the graphs obtained by using the extensional constraint representation with allowed and disallowed assignments, respectively. Both graphs have 1 constraint node (coloured grey), 2 variable nodes of the same colour (here represented using shape as square), 8 value nodes of the same colour (white), and either 10 or 6 assignment nodes, respectively, of the same colour (here represented using size and colour as small and

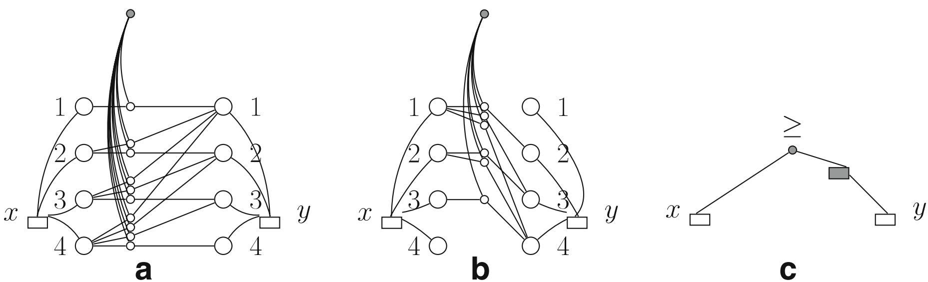


Fig. 2 Allowed extensional, disallowed extensional, and intensional graphs

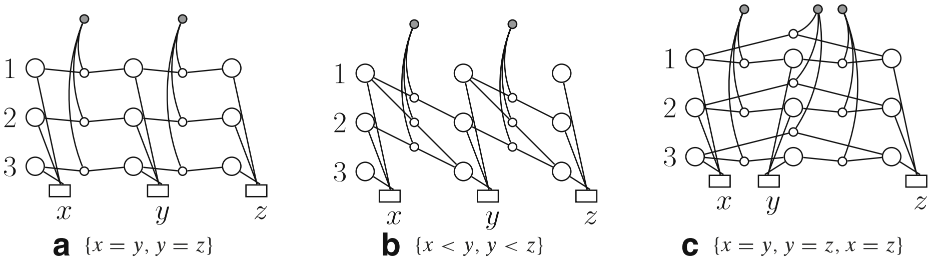


Fig. 3 Graphs for CSPs = ($\{x, y, z\}$,$\{1, 2, 3\}$,C) where C is:

white). Figure 2c shows the graph obtained for the same CSP using the intensional constraint representation. This graph needs 1 constraint node and 2 variable nodes as before, plus a dummy node (represented as a grey square) to break the symmetry.

Example 4 Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x = y, y = z\})$. Figure 3a shows the graph obtained by using the extensional constraint representation with allowed assignments. The graph has three value nodes per variable.[1] It also has three assignment nodes corresponding to each allowed assignment $\{x = 1, y = 1\}$, $\{x = 2, y = 2\}$, $\{x = 3, y = 3\}$ of $x = y$, and another three for those of $y = z$. All assignment nodes have the same colour (small and white). Finally, the graph has two constraint nodes, each connected to its associated assignment nodes and mapped to the same colour (grey), since they represent constraints of the same kind (equality). The graph associated with CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y, y < z\})$ using the extensional constraint representation with allowed assignments is shown in Fig. 3b.

While the method described above is correct (any automorphism of the graph corresponds to a symmetry of the CSP), it is not complete (some CSP symmetries might not appear in the graph). This was demonstrated by Puget [13] using the graph of Fig. 3a which represents constraints $\{x = y, y = z\}$. The graph does not contain any variable symmetries involving y, even though both $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$ and $\langle y, z\rangle \leftrightarrow \langle z, y\rangle$ are symmetries of the CSP. To reduce this problem, Puget suggests to take the transitive closure of equality and $\leq$ constraints, and also to replace constraints such as $x \leq y$ and $y \leq x$ by $x = y$. Applying this to the graph of Fig. 3a leads to the addition of $x = z$, and results in the graph of Fig. 3c, which does contain symmetries $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$ and $\langle y, z\rangle \leftrightarrow \langle z, y\rangle$. Note, however, that the transitive closure does not make the method complete.

2.3 Puget's representation using Boolean variables

As mentioned before, there are symmetries of the CSP that cannot be expressed as the composition of variable and value symmetries present in the CSP. This is the case, for instance, for the rotational symmetries of the n-queens problem. These non-compositional symmetries cannot be expressed using the intensional representation

[1] Note that, for simplicity, only the leftmost value nodes are labelled, their associated value being shared by all value nodes at the same horizontal level.

of constraints proposed by Puget since, as he indicates, it is only suitable for variable symmetries. Neither can they be represented using the intensional representation, due to the existence of variable nodes and to the different colour used for different kinds of constraints.

Puget addresses this issue by proposing a new representation of CSPs, one that uses Boolean variables instead of finite domain variables. There is a standard mapping of finite domain CSPs to the Boolean representation by introducing a Boolean variable for each literal in the original CSP. An allowed assignment of a constraint in the original CSP corresponds to an assignment of *true* to all the Boolean variables representing literals in the assignment. A solution to the original CSP corresponds to an instantiation of the Boolean variables such that:

- precisely one Boolean variable for each original variable is set to *true*
- each original constraint has (at least one) allowed assignment

Puget proposes a graphical representation of the original CSP based on this Boolean model (actually, using zero-one rather than Boolean variables). He employs an intensional representation of the Boolean model, together with a node for each original constraint, linked to the Boolean constraint representing each allowed assignment of the original constraint (illustrated in Fig. 6 of Puget [13]). As a result, there are no longer variable nodes and, thus, some non-compositional variable-value symmetries can be represented.

However, automorphisms of this graph representation do not necessarily correspond to solution symmetries of the original CSP. A counter example can be illustrated with the CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y\})$, (later shown using a different graphical representation in Fig. 13), under the permutation $\langle x = 3\rangle \leftrightarrow \langle z = c\rangle$, for any $c \in \{1, 2, 3\}$, that is not a symmetry of the CSP. In Lemma 2 below, we impose some restrictions on the Boolean representation to guarantee that graph automorphisms do indeed correspond to symmetries.

Moreover, the link to the original constraint node precludes symmetries involving different constraints (such as an *all_different* and a disequation). Interestingly, Puget's Boolean model for the *all_different* constraint does not appear to include a link back to the original constraint node. For clarity, we briefly summarise four alternative representations for this constraint under Puget's approach.

The first representation is an intensional representation of the original constraint. This comprises a node for the constraint, a node for each variable in its scope, and an edge between the constraint node and each variable node (Fig. 4a). The second representation is the extensional representation of the original constraint with $O(m^n)$ nodes representing allowed assignments of n variables each with a domain size of m (Fig. 4b). A third representation is the standard Boolean one with an assignment node for each allowed assignment. This is almost the same as the extensional representation, but without any nodes corresponding to the original variables (Fig. 4c). The fourth representation is another Boolean one, but this time without any nodes representing the original constraints. The *all_different* constraint over n variables v_i can be represented by m constraints (one per value, c_j), each of which is connected to the n Boolean variables $b_{ij} : i \in 1..n$, representing $v_i = c_j$. Each Boolean constraint states that only one of the connected Booleans is true (Fig. 4d).

The reason why constraint nodes are needed in the third representation and not in the fourth representation of the *all_different* constraint is as follows: while the (many)

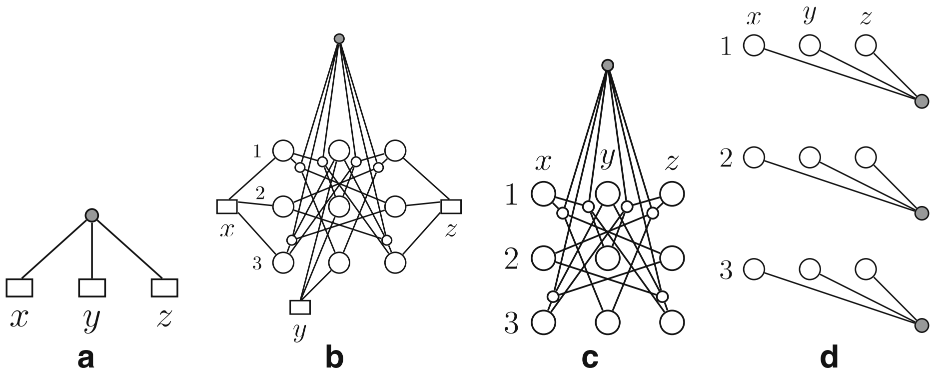


Fig. 4 CSP = ({x, y, z},{1, 2, 3}, {*all_different*({x, y, z})})

assignments corresponding to an original constraint in the third representation form a logical *disjunction* of the Boolean constraints, in the fourth representation they form a logical *conjunction*. The constraint node in the third representation removes any incorrect symmetries between a disjunction of assignments and a conjunction of constraints. We explore the consequences of this issue further in the next section.

The resulting graph (Fig. 4b shows an example for $n = 3$) is much smaller ($O(m.n)$ nodes) than a graph that uses the extensional form of the original *all_different* constraint which requires ($O(m^n)$ nodes).

To complement this representation, Puget also proposes a similar *all_different* constraint between the different values of a variable, connecting the Boolean nodes $b_{ij} : j \in 1..m$. This is necessary to ensure graph automorphisms correspond to solution symmetries. Indeed, these constraints are not only necessary, but also sufficient, as shown in the next section. However, if this is expressed via an *all_different* constraint, then opportunities for detecting symmetries may be lost since this representation does not allow a disequation between two values to participate in a symmetry with a disequation between two variables (a similar situation is shown in Example 12).

Note that, as shown in Section 4.1, an extensional representation over Boolean variables can also stay within the same $O(m.n)$ number of nodes. However, both the $O(m.n)$ Boolean representations of *all_different* may lose symmetries due to the interaction between this (very special kind of) representation and that of other constraints in the CSP.

2.4 Puget's representation for expressions

Puget's method is based on constraints whose arguments are distinct variables. In order to be able to handle constraints involving expressions, Puget proposes to represent any expression of the form x_i *op* x_j, where $x_i, x_j \in X$ are distinct variables, as the extensional constraint associated with $op(x_i, x_j, t)$, where t is a new (temporary) variable which is then used to replace the expression x_i *op* x_j as the constraint's argument (see Fig. 5). We believe this method was suggested because (a) it reuses the already defined constraint representation, and (b) uses the same colour for constraints with and without complex expressions. For example, $A < B$ can be

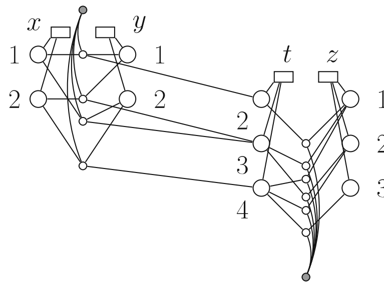


Fig. 5 Graph of CSP $(\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1, 2, 3\}\}, \{x + y > z\})$. The extra variable t represents $x + y$, with domain $D_t = \{2, 3, 4\}$

represented by the same colour as $C < D + 1$ if the latter constraint is expressed as a combination of $E = D + 1$ and $C < E$. This reduces the syntax dependency of the graph and thus might result in more symmetries being detected.

Since this approach can lead to very large graphs, Puget also proposed an alternative approach for handling expressions of the form $op(x)$, where the variable x is only allowed to occur once in the expression. The idea then is to use the literal $x = d$ to represent $t = op(d)$ wherever it would have occurred (see Fig. 6). Although this representation results in compact graphs, it is only suitable for some constraints.

Expressions with more than one variable are, therefore, broken into sub-expressions, each of which is represented by a new (temporary) variable t_i. Constraints involving expressions as arguments are simply treated by replacing each such argument by a new variable representing the expression. As noted by Puget and others [13, 15], this can lead to the unintentional loss of symmetries due, for example, to the associative nature of operators.

Example 5 Consider the constraint $x + y + z > w$. If the constraint is parsed as $(x + y) + z > w$, it would be transformed into constraint $t_1 > w$, where the new variable t_1 has associated constraint $+(t_2, z, t_1)$ (representing the constraint $t_2 + z = t_1$), and new variable t_2 has associated constraint $+(x, y, t_2)$. Although in the expression all three variables are interchangeable, the associated graph only has the variable symmetry $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$.

The problem can be ameliorated [13, 15] by representing multiple occurrences of a binary associative operator with a single n-ary operator. For instance, in the previous example we would only introduce one extra variable t_3, and the constraint $+(x, y, z, t_3)$. As recommended in [13], non-symmetric binary arithmetic operations, such as $x - y$ and x/y, are decomposed using their unary inverse operators, resulting in $x + (-y)$ and $x * (1/y)$. This allows further grouping of associative operators while at the same time preventing the creation of false symmetries. Unfortunately, as

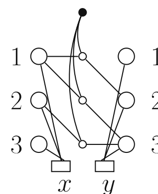


Fig. 6 Graph of CSP $(\{x, y\}, \{1, 2, 3\}, \{x + 1 > y\})$. Node $x = 1$ represents $2 = +1(1)$, and $x = 2$ represents $3 = +1(2)$

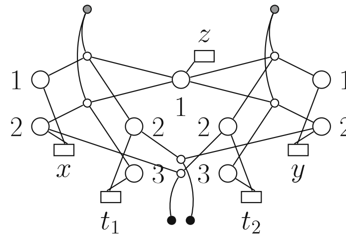


Fig. 7 Graph of CSP $(\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1\}\}, \{x + z \neq y, y + z \neq x\})$

mentioned before, this preprocessing only reduces the problem instead of eliminating it, since the intermediate variables can still prevent some symmetries from being captured by the graph, even after performing all the preprocessing steps indicated above.

Example 6 Consider the CSP $(\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1\}\}, \{x + z \neq y, y + z \neq x\})$. This CSP has a variable symmetry $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$, and a value symmetry $\langle 1, 2\rangle \leftrightarrow \langle 2, 1\rangle$, for both x and y. The expressions in the two constraints are represented by $t_1 = x + z$ and $t_2 = y + z$. The associated extensional graph is shown in Fig. 7, with grey and black constraint nodes linking assignment nodes for equality and disequality constraints, respectively. It can be seen that the graph captures the variable symmetry (achieved by reflecting the graph in a vertical axis positioned over value node 1 of z), but not the value symmetry (which would be achieved by reflecting the graph in a horizontal axis positioned in between value nodes 1 and 2 of x and y).

The above discussion highlights the considerable influence that the constraint syntax bears on the resulting graph. Since the same constraint can usually be expressed in several equivalent ways, it would seem advantageous to determine which normal form would yield a graph that captures the greatest number of symmetries. It was while trying to determine such a normal form that we decided to abandon the above method, since it not only generated too many intermediate variables (as already indicated by Puget), but it could also easily result in symmetries being missed due to the use of a particular syntax.

2.5 Representing sets

Sets are used in modelling many problems—arguably most problems coming from the real-world. Consider a set variable $x_i \in X$ known to be a subset of set S. Its domain D_i is equal to the power-set of S, i.e., $D_i = \{S' | S' \subseteq S\}$. In any graph representation which includes value, or literal, nodes, a node is therefore required for every element of the power set, and this leads to large graphs. Using the extensional method (c.f. Fig. 4b) the graph has one variable node and $2^{|S|}$ value nodes (e.g., if $S = \{1, 2, 3, 4\}$ the graph contains 1 variable node for x_i and 16 value nodes representing values $\{\}, \{1\}, \{2\}, \ldots, \{2, 3, 4\}$, and $\{1, 2, 3, 4\}$). Set constraints are then extensionally represented as usual by using their allowed or disallowed assignments.

An alternative approach is to use a Boolean representation, (c.f. Fig. 4c). Then, rather than using each element in the powerset of S to create a value node, we would use each element in the set, i.e., we would obtain $|S| + 1$ nodes where one node represents the *empty* set, and $\forall d \in S$ there is a node representing $d_i \in x_i$ (e.g., if

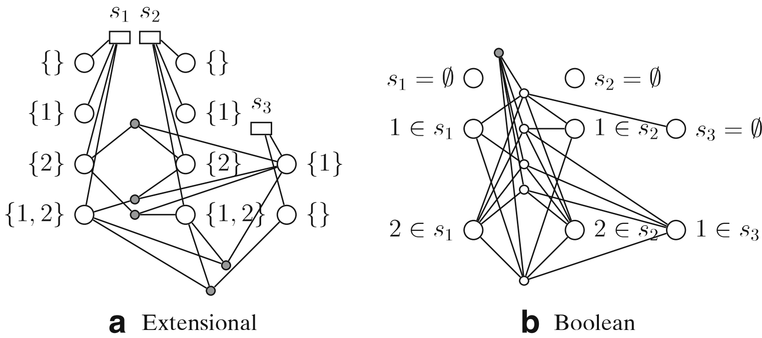


Fig. 8 Different representations for set constraint $|(s_1 \cap s_2) \cup s_3| = 2$

$S = \{1, 2, 3, 4\}$ the graph contains 5 nodes representing $x_i = \{\}$, $1 \in x_i$, $2 \in x_i$, $3 \in x_i$, and $4 \in x_i$). A constraint on the set is represented by its allowed assignments. A set is represented, in an assignment, by its elements. Thus the assignment includes all the elements which belong to the allowed set, (or the empty set if there are none). The constraint is, as usual, the *disjunction* of the represented assignments. In the graph, a constraint is represented by a constraint node and a node for each of its allowed assignments. The assignment node is then linked to each of the nodes representing an element in the allowed set, or the node representing the empty set, if there are none. This Boolean representation results in fewer nodes but more edges.

The significance of these alternative representations for our implementation is shown in Section 5.1 below.

Example 7 Figure 8a, b show the two alternative graphs obtained for CSP $(\{s_1, s_2, s_3\}, D, \{|(s_1 \cap s_2) \cup s_3| = 2\})$ where $D_{s_1} = D_{s_2} = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$, and $D_{s_3} = \{\{\}, \{1\}\}$.

3 A new graph representation

3.1 Allowed and disallowed assignments

Instead of using intensional constraints or factoring out expressions with temporary variables, we believe it is cleaner and simpler to return to extensional constraints. An important motivation for us is to eliminate different "kinds" of constraints, which have different colours and stick to just two kinds: constraints represented extensionally by *allowed* and by *disallowed* assignments. When seen in this light, it becomes clear that we can avoid the representation of temporary variables and constants by absorbing expressions into the constraint in which they appear. We also decided to drop variable nodes and, instead, use literal nodes rather than value nodes. The consequences of this will be discussed in Section 3.3 below.

Example 8 Consider the CSP $(\{x, y, z\}, D, \{x + y > z\},)$ where $D_x = D_y = \{1, 2\}$ and $D_z = \{1, 2, 3\}$. The ternary constraint $x + y > z$ can simply be represented extensionally by its set of allowed assignments, $\{\{x = 1, y = 1, z = 1\}, \{x = 1,$

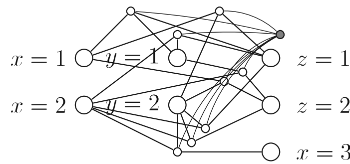


Fig. 9 Representing expressions as allowed assignments

$y = 2, z = 1\}, \{x = 1, y = 2, z = 2\}, \{x = 2, y = 1, z = 1\}, \{x = 2, y = 1, z = 2\}, \{x = 2, y = 2, z = 1\}, \{x = 2, y = 2, z = 2\}, \{x = 2, y = 2, z = 3\}\}$, all linked to an additional constraint node, as illustrated by Fig. 9.

We would like to simplify the graph further by eliminating the constraint nodes. This, however, cannot be achieved if the CSP contains at least two constraints $c_1, c_2 \in C$ such that $c_1 \neq c_2$ and $vars(c_1) = vars(c_2)$. This is because while each constraint must be interpreted as the union of its allowed assignments, their conjunction must be interpreted as the intersection of the set of assignments allowed by each. Without constraint nodes, the graph cannot distinguish between the set of assignment nodes representing $c_1 \wedge c_2$ and that representing $c_1 \vee c_2$. By contrast, representing $c_1 \wedge c_2$ by their disallowed assignments is correct and unambiguous even without constraint nodes. This is because the set of disallowed assignments $\{A_{11}, \ldots, A_{1s}\}$ and $\{A_{21}, \ldots, A_{2t}\}$ for c_1 and c_2, respectively, is interpreted as the conjunction of all their disallowed assignments, i.e., $\neg A_{11} \wedge \ldots \neg A_{1s} \wedge \neg A_{21} \wedge \ldots \neg A_{2t}$.

Example 9 Figure 10a, b show the graphs obtained by representing the disallowed assignments of the constraints in CSPs (X, D, C) and (X, D, C'), respectively, where $X = \{x, y\}$, $D_x = D_y = \{1, 2\}$, $C = \{x > y, x < y\}$, and $C' = \{x \neq y\}$. It is clear that, without the constraint nodes, the graphical representation of these two CSPs are indistinguishable, even though while the first CSP has no solutions, the second has two. The confusion is due to $x \neq y$ being logically equivalent to $(x > y \vee x < y)$. Figure 10c, d show the graphs obtained by representing the disallowed assignments of the constraints (note that, for clarity, identical assignments disallowed by different constrains have been merged). As it is clear from the figure, the graphs are perfectly distinguishable even though constraint nodes are not represented.

The simplicity of this method is pleasing since it eliminates problems such as the explicit representation of constants (which is now avoided regardless of whether the constant appears as a constraint argument or not), the normalisation required when multiple occurrences of a variable appeared in a constraint (such as A×A=1, which can now be easily treated by computing the values of A for which the constraint is satisfied), the problem of associative and non-symmetric operands appearing in the same constraint, and in general, any such normalisation issue affecting a single constraint. For instance, Fig. 11 shows the graph obtained for the CSP of Example 6 using the new representation method.[2] The elimination of temporary variables yields a graph that has not only the variable symmetry $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$, but also the value symmetry $\langle 1, 2\rangle \leftrightarrow \langle 2, 1\rangle$ for both x and y.

[2]Note that, for simplicity, only the leftmost (topmost) literal nodes are labelled with the associated value (variable), which is shared by all literal nodes at the same horizontal (vertical) level.

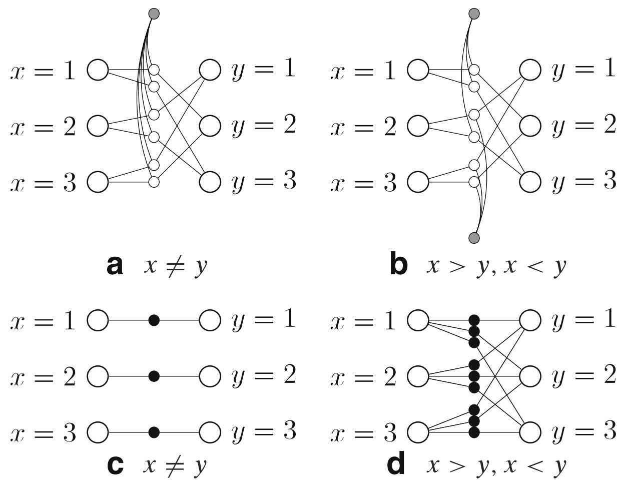


Fig. 10 Graphs using allowed (**a**, **b**) and disallowed (**c**, **d**) assignments

The method also avoids syntactical issues regarding constraints whose name appears to be different but is actually equivalent (e.g., $x < y$ and $z > w$ should be considered as constraints of the same kind). However, some problems remain when a CSP has two constraints with identical scopes (see Section 3.3).

3.2 Disallowed assignments and the microstructure complement

Given the above discussion, it would seem advantageous to represent a CSP using only its disallowed assignments. Let us consider the advantages and disadvantages of such an approach.

Definition 1 A CSP (X, D, C) is represented by the *disallowed assignments graph* if the graph contains two kinds of nodes:

- *Literal nodes,* each representing the assignment of a specific value to a specific variable
- *Assignment nodes,* each representing either a disallowed assignment from any constraint in C, or a (disallowed) pair of distinct literals $\{x = a, x = b\}$ for all $x \in X$ and all $a, b \in D_x$ such that $a \neq b$

All literal nodes have one colour and all assignment nodes have another. The graph contains an edge from each assignment node to each of the literals involved.

Example 10 Figure 12 illustrates the disallowed assignments graph for the three- and four-queens problems (with disallowed disequality assignments shown as small black nodes). Note the need to represent the disallowed pairs of distinct literals

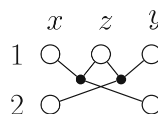


Fig. 11 Graph for CSP $(\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1\}\}, \{x + z \neq y, y + z \neq x\})$

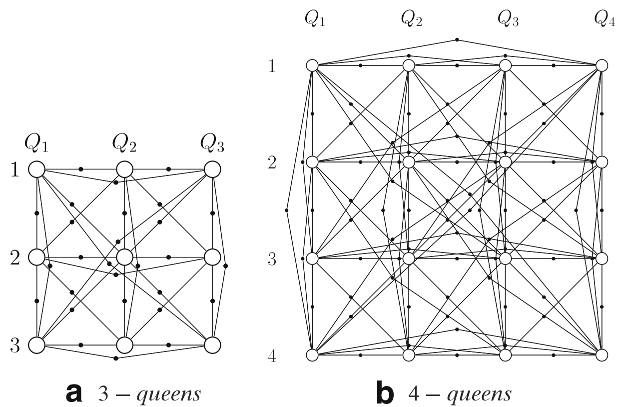


Fig. 12 Graph for instances of N-queens using disallowed tuples

for each variable, to capture all the symmetries of the chessboard. While they are also captured by Puget's Boolean representation, the variable nodes do not allow Puget's extensional representation to capture the non-compositional variable-value symmetries that indicate a rotation of the chessboard.

The disallowed assignments graph is similar to the microstructure complement introduced by Jégou [9] and extended by Cohen et al. [2]. Given a CSP (X, D, C), its *microstructure* is a hypergraph with a node for each literal $x_i = d_i$ such that $x_i \in X, d_i \in D_i$, and a hyperedge for every assignment allowed either by a specific constraint, or by the lack of a constraint between the variables involved. The microstructure *complement* is the complement of this graph; i.e., the hyperedges represent assignments that are *not* allowed either by a constraint or by the fact that the literals belong to the same variable.

The only difference between the disallowed assignments graph and the microstructure complement is that each disallowed assignment is represented by an assignment node and the literals linked to it, whilst the microstructure complement represents a disallowed assignment by a single hyperedge linking the literals.

The main drawback of both the microstructure complement and the disallowed assignments graph is their size. Firstly, they have, in general, more nodes than necessary. Every value for every variable is represented as a node, although many of these nodes could never appear in a solution (techniques for pruning nodes will be discussed below). And secondly, the number of hyperedges (or assignment nodes) in the graph is high. A mathematical equation, such as $x = 2y + z$ requires approximately d^3 hyperedges, where d is the size of the domains of x, y and z. While for some constraints the set of disallowed assignments is the most compact way to represent the constraint, many others, such as mathematical equations, are much more compactly represented by their allowed assignments. Moreover, the microstructure complement requires d^2 edges to disallow multiple assignments for a variable in the CSP with domain size d. Thus for n variables, nd^2 edges are needed. While this number could be kept to nd using intensional constraints, as described before, this would limit the symmetry possibilities, making non-compositional variable-value symmetries unlikely. Therefore, in the next section we consider an alternative graph in which only allowed assignments are represented explicitly.

3.3 Allowed assignments and the microstructure

The definition of the microstructure provided in the previous section suffers from a flaw caused by the same reason that prevented us in Section 3.1 from eliminating constraint nodes for allowed assignments: if two or more constraints have the same scope, then the set of hyperedges over that scope represents the disjunction, instead of the conjunction, of the constraints. This flaw can be easily fixed, however, by a preprocessing step which replaces each set of constraints that have the same scope by a new constraint whose allowed assignments are those that satisfy all constraints. From now on, we will assume that every CSP (X, D, C) has already been preprocessed and, therefore, it is true that for every two distinct constraints $c_1, c_2 \in C : vars(c_1) \neq vars(c_2)$.

Unfortunately, there is another serious drawback to using the microstructure: the inclusion of a hyperedge for each assignment "allowed because there is no constraint between the associated variables". Assuming there are n variables in the CSP (X, D, C), there will be 2^n subsets of X, with each subset X_i being either equal to $vars(c)$ for some $c \in C$, or unconstrained. An unconstrained set of variables $\{x_i, \cdots, x_j\}$ has $|D_{x_i}| \times \ldots \times |D_{x_j}|$ allowed assignments. Since the number of constraints is typically much smaller than 2^n, the number of hyperedges in the microstructure is typically very large indeed.

We seek a graphical representation of the CSP that has a small number of edges, but for which graph automorphisms correspond to solution symmetries. Luckily, it turns out not to be necessary to add allowed assignments for *every* set of variables that do not form the scope of a constraint in the CSP. It is sufficient to add allowed assignments for each pair of distinct variables which do not both belong to the scope of a constraint.

If (X, D, C) is a CSP, its *binary constraint completion*, BC, is the set of binary constraints whose scopes are the pairs of distinct variables $x_i, x_j \in X$ for which there is no constraint $c \in C$ with $\{x_i, x_j\} \subseteq vars(c)$ - i.e., the constraints in BC are logically equivalent to *true*.

Definition 2 A CSP (X, D, C) is represented by the *allowed assignments graph* if the graph contains two kinds of nodes:

- *Literal nodes,* each representing the assignment of a specific value to a specific variable
- *Assignment nodes,* each representing an allowed assignment from a constraint in $C \cup BC$.

All literal nodes have one colour and all assignment nodes have another. The graph contains an edge from each assignment node to each of the literals involved.

Lemma 1 *Every automorphism f of an allowed assignments graph for CSP (X, D, C) represents a solution symmetry.*

Proof Let S be a solution to the CSP. We will prove that $f(S)$ is also a solution by first showing it is a complete assignment, and then showing it satisfies each constraint in C.

Let lit_1 and lit_2 be two distinct literals in S. Then, $\{var(lit_1), var(lit_2)\} \subseteq vars(c)$ for some $c \in C \cup BC$. Since S is a solution, it satisfies c and, therefore, $\{lit_1, lit_2\} \subseteq A$ for some assignment A allowed by c. As a result, they must both be linked to at least one allowed assignment node n. By the definition of automorphism, $\{f(lit_1), f(lit_2)\}$ are linked to $f(n)$, which means they also belong to an assignment allowed by some constraint $c' \in C \cup BC$. By the definition of an allowed assignment, $var(lit_1) \neq var(lit_2)$ and $var(f(lit_1)) \neq var(f(lit_2))$. Since this holds for every pair of literals in S and $f(S)$, we have that $card(f(S)) = card(S) = card(X)$, and we have shown that $f(S)$ is a complete assignment.

Let us now show that every constraint in C is satisfied by $f(S)$. If there are m constraints in C, then there are m subsets of S which correspond to allowed assignments. Let $c_1, c_2 \in C$ be two different constraints and $A_1, A_2 \subseteq S$ be assignments allowed by c_1 and c_2, respectively. By assumption of the preprocessing step, $vars(c_1) \neq vars(c_2)$. Also, by definition of automorphism, the image $f(A)$ of any allowed assignment A is also an allowed assignment and, therefore, for every two distinct literals $lit_1, lit_2 \in A$ we have $var(lit_1) \neq var(lit_2)$ and $var(f(lit_1)) \neq var(f(lit_2))$. Therefore, if $v \in vars(c_1) \setminus vars(c_2)$, and $v = var(lit)$ with $lit \in A_1$, then $var(f(lit))$ is not in the set of variables over which $f(A_2)$ is an assignment. It follows that $f(A_1)$ and $f(A_2)$ are allowed assignments over distinct sets of variables and, therefore, they belong to two different constraints. This means that $f(S)$ also satisfies m distinct constraints and, therefore, all constraints in C. Since $f(S)$ is a complete assignment, it must also be a solution. □

Figure 13 shows a CSP where, if the BC is excluded, the graph has automorphisms that are not solution symmetries.

3.4 A graph including allowed and disallowed assignments

We now present a new graph representation for CSPs that does not require all constraints to use the allowed (or disallowed) assignments and, thus, permits different constraints to use different assignments. The representation takes many ideas from the work of Puget [13] but is also closely related to the microstructure and microstructure complement of Cohen et al. [2]. As before, our graph representation of a CSP (X, D, C) has a node for every literal (and, thus, for every value of the domain of every variable in X). However, we now admit both allowed assignments and disallowed ones, distinguished by different colours.

We call an *allowed* constraint one that is represented by all its allowed assignments, and a *disallowed* constraint one represented by all its disallowed assignments.

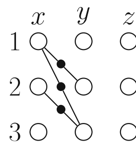


Fig. 13 CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y\})$. The graph has the automorphism $\langle x = 3\rangle \leftrightarrow \langle z = 3\rangle$, which is not a solution symmetry

Definition 3 A CSP (X, D, C) is represented by the *full assignments graph* if the graph contains three kinds of nodes:

- *Literal nodes,* each representing the assignment of a specific value to a specific variable
- *Allowed assignment nodes,* representing an allowed assignment from a constraint in C
- *Disallowed assignment nodes,* either representing a disallowed assignment from a constraint in C, or a (disallowed) pair of distinct literals $\{x = a, x = b\}$ for all $x \in X$ and all $a, b \in D_x$ such that $a \neq b$.

All literal nodes have one colour, all allowed assignment nodes have another, and all disallowed assignment nodes have a third. The graph contains an edge from each assignment node to each of the literals involved.

Two conditions are imposed to ensure that graph automorphisms correspond to solution symmetries.

1. Each constraint must be either allowed or disallowed
2. Either:
   - every pair of variables is in the scope of an allowed constraint (i.e., $\forall x, y \in X, x \neq y : \exists c \in C, x, y \in vars(c)$), or
   - every pair of literals within a variable is linked by disallowed assignments, for all variables (i.e., $\forall x \in X, \forall a, b \in D_x, a \neq b$: $\exists$ an assignment node linking literal nodes $x = a$ and $x = b$)

Lemma 2 *Every automorphism f of the full assignments graph for CSP (X, D, C) represents a solution symmetry.*

Proof Let S be a solution to the CSP. We will prove that $f(S)$ is also a solution by first showing it is a complete assignment, and then showing it satisfies each constraint in C.

If every pair of variables is in the scope of an allowed constraint, then this is proved in Lemma 1 above. Otherwise, for each variable, all its pairs of literals are linked by disallowed assignments. Let us reason by contradiction and assume that $f(S)$ is not a complete assignment. Then, there must be two literals, say $f(lit_1)$ and $f(lit_2)$, for the same variable. Therefore, $\{f(lit_1), f(lit_2)\}$ must be linked to one binary disallowed assignment node and, by the definition of automorphism, $\{lit_1, lit_2\}$ must also be linked to a binary disallowed assignment node. But this is impossible since they belong to a solution. We conclude that $f(S)$ must be a complete assignment.

Let us now show that every constraint in C is satisfied by $f(S)$. If there are m constraints in C, represented by allowed assignments, then it follows, as in the proof of Lemma 1 above, that $f(S)$ also includes allowed assignments from m different constraints, and therefore $f(S)$ satisfies all the constraints in C represented by allowed assignments. Otherwise, if $f(S)$ includes a set of literals linked to a disallowed assignment, then so must S have, which would contradict S being a solution. We conclude that $f(S)$ is a complete assignment that satisfies all the constraints and, therefore, it is a solution. □

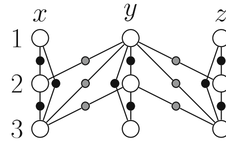


Fig. 14 Full assignments graph of CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y, y < z\})$

Lemma 3 *Not every solution symmetry for a CSP (X, D, C) is an automorphism f of its full assignments graph.*

Proof It is easy to prove the lemma by contradiction. Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y, y < z\})$ whose only solution is $\{x = 1, y = 2, z = 3\}$. While this CSP has the solution symmetry $\langle x = 2, x = 3\rangle \leftrightarrow \langle x = 3, x = 2\rangle$, this is not a constraint symmetry since it maps the allowed assignment $\{x = 2, y = 3\}$ of constraint $x < y$ to the disallowed assignment $\{x = 3, y = 3\}$. It is clear from its full assignment graph (Fig. 14) that the solution symmetry is not an automorphism of the graph either. □

While supporting both allowed and disallowed assignments makes it possible to reduce the size of the graph, it might lead to a loss of constraint symmetries. This can occur even if we consistently represent constraints using the method that ensures the minimum number of assignments.

Example 11 Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, C)$, where $C = \{c_1(z, y), c_2(x, y), x = z\}$, $c_1 = \{1, 2\} \times \{1, 2, 3\}$ and $c_2 = \{1\} \times \{1, 2, 3\}$. This CSP contains a solution symmetry, $\langle x = 1\rangle \leftrightarrow \langle z = 1\rangle$. As illustrated by Fig. 15, if this CSP is represented using only disallowed assignments or only allowed assignments, the symmetry is present in the graph. However, if we use the representation with the smallest graph (disallowed assignments for $c_1(z, y)$ and allowed assignments for $c_2(x, y)$ and $x = z$), then that symmetry is not present in the graph.

Lemma 4 *All results proven for a full assignments graph hold for the allowed and disallowed assignments graphs.*

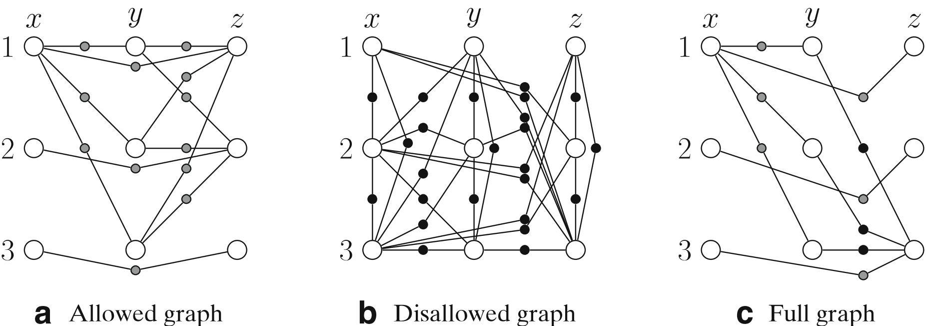


Fig. 15 Symmetry $\langle x = 1\rangle \leftrightarrow \langle z = 1\rangle$ is not present in (c) (see Example 11)

Proof Immediate since, by definition, any allowed (disallowed) assignments graph is an instance of a full assignments graph in which only allowed (disallowed) assignments are used for representing constraints. □

4 Reducing graph size

While the full assignment graph might result in smaller graphs than those obtained using only allowed or only disallowed assignments, even the full assignments graphs tend to be rather large (see Table 1 for size data). For example, the number of nodes in the full assignment graph of a CSP is the sum of:

- the number of literals, which is the product of the variable domain sizes
- the number of allowed assignments for constraints and its binary constraint completion, if these are explicitly represented

Table 1 Graph sizes

Instance	Graph details						
	Puget's (Ext)		Puget's (Bool)		Ours		
	Nodes	Edges	Nodes	Edges	Nodes	Edges	Gens
bibd-3-3-1-1-0	216	477	216	477	**141**	**297**	4
bibd-6-10-5-3-2	23,737	243,576	23,737	243,576	**3,857**	**26,730**	14
golf-2-2-2	1,722	4,670	1,722	4,670	**1,034**	**2,640**	6
golf-2-2-3	25,242	73,854	25,242	73,854	**10,650**	**29,344**	8
golf-2-3-2	62,841	184,515	62,841	184,515	**24,762**	**68,109**	10
golf-3-2-2	4,245	11,667	4,245	11,667	**2,703**	**7,374**	8
golomb-4	1,245	3,332	**1,006**	**2,708**	2,484	5,456	1
golomb-5	4,815	13,505	**3,670**	**10,380**	9,380	21,250	1
golomb-6	14,658	42,072	**10,809**	**31,272**	27,978	64,422	1
golomb-7	37,632	109,424	**27,181**	**79,583**	70,665	164,297	1
graceful-3-2	1,626	4,380	**1,085**	**3,000**	2,235	5,070	4
graceful-5-2	27,160	78,520	**17,897**	**52,520**	38,155	91,390	5
latin-10	11,000	28,000	**1,300**	**3,000**	14,500	27,000	27
latin-11	15,972	41,261	**1,694**	**3,993**	21,296	39,930	30
latin-12	22,464	58,752	**2,160**	**5,184**	30,240	57,024	33
latin-13	30,758	81,289	**2,704**	**6,591**	41,743	79,092	36
latin-14	41,160	109,760	**3,332**	**8,232**	56,252	107,016	39
mostperfect-4	–	–	–	–	**80,704**	**314,112**	5
nnqueens-4	460	976	**152**	**304**	464	800	5
nnqueens-5	1,110	2,525	**270**	**605**	1,175	2,100	6
nnqueens-6	2,282	5,436	**432**	**1,056**	2,496	4,560	7
queens-10	1,265	3,160	**154**	**396**	1,570	2,940	2
queens-20	9,730	26,620	**514**	**1,596**	12,940	25,080	2
queens-30	32,395	91,380	**1,074**	**3,596**	44,110	86,420	2
queens-40	76,260	218,440	**1,834**	**6,396**	105,080	206,960	2
steiner-5	3,282	9,276	3,282	9,276	**2,115**	**5,904**	6
steiner-6	41,975	123,090	41,975	123,090	**22,440**	**65,850**	9
steiner-7	347,760	1,032,689	347,760	1,032,689	**154,294**	**459,557**	12

- the number of disallowed assignments for constraints, and for pairs of literals from the same variable, if these are explicitly represented

The following subsections describe some general methods that help reduce the size of the graph.

4.1 Minimising the number of assignment nodes

In the worst case, a constraint over k variables may need to be represented in the full assignment graph by $O(d^k)$ allowed or disallowed assignment nodes, where d is the size of the smallest domain. A way to minimise the number of assignment nodes is to keep k as small as possible. Consider for example an *all_different* constraint on k variables, where d is larger than k. Using allowed assignments the graph representation requires $d \times (d-1) \times \ldots \times (d-k-1)$ assignment nodes or $O(d^k)$. Using disallowed constraints the number is $O(d^{(k-1)})$. However, when split into $k \times (k-1)$ binary constraints, the total number of disallowed assignments is d per binary constraint, making a total of $O(k^2 \times d)$.

Breaking down each constraint into a logically equivalent conjunction of constraints with as small scope as possible, has a very useful side-effect: it will tend to increase the number of constraints with the same scope, which can be integrated (during the preprocessing step) into a single constraint, thereby increasing the number of detected symmetries.

Example 12 Consider the CSP $(\{x_1, x_2, x_3, x_4\}, \{1, \ldots, N\}, \{all_different(X),\ x_2 \neq x_4\})$, where *all_different* is represented using N disallowed assignments nodes. Then, the representation of $x_2 \neq x_4$ will limit the symmetries of the graph to those generated by $\langle x_1, x_3\rangle \leftrightarrow \langle x_3, x_1\rangle$ and $\langle x_2, x_4\rangle \leftrightarrow \langle x_4, x_2\rangle$. Thus, symmetries such as $\langle x_1, x_2\rangle \leftrightarrow \langle x_2, x_1\rangle$ will be lost since they do not correspond to automorphisms of the associated graph. However, if *all_different* is represented by the disallowed assignments of the equivalent constraint $\{x_i \neq x_j | 1 \leq i < j \leq 4\}$, all variable symmetries are present in the graph. Figure 16 shows the graphs associated with the two representation of the CSP for $N = 1$.

Note that we can also represent the *all_different* constraint extensionally using only $O(d.k)$ nodes. Suppose we order the k variables in the constraint, $v_1, \ldots, v_k$. We will use Boolean variables b_{jc} to represent $v_j = c$. We now introduce, for each value c, k new Boolean variables $t_{jc} : j \in 1..k$. t_{jc} is *true* (1) if any of $b_{1c}, \ldots, b_{jc}$ are *true*. $t_{1c} = b_{1c}$ and for each $j \in 2..k$ there is a constraint with scope $t_{j-1,c}, b_{j,c}, t_{j,c}$ defined by three allowed assignments: $< 1, 0, 1 >$, $< 0, 1, 1 >$, $< 0, 0, 0 >$. There are just k such constraints for each value c, and each constraint has just three assignments, so the total number of edges required is $3.d.k$ which is $O(d.k)$. While this representation is also very compact, it has the same problem as the Boolean representation proposed by Puget: it might lead to a loss of symmetry detection if it has to interact with

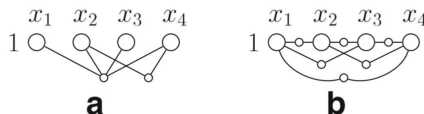


Fig. 16 Representing *all_different* using n-ary and binary constraints

that obtained for other kinds of constraints. We have therefore not used it in our implementation.

If all constraints represented in the graph have less than k variables in their scope, then the number of edges is less than n^k where n is the number of nodes. Moreover, as pointed out in by Cohen et al. [2], by adding enough disallowed assignments over k variables it is possible to create a (microstructure complement) graph whose automorphisms represent all solution symmetries of the CSP. Their proof is easily adapted to show the same holds true of the full assignment graph.

This result gives an upper bound on the size of the graph needed to capture all the solution symmetries of a given CSP. It follows that representing constraints by an equivalent conjunction of constraints of minimum possible arity, we achieve a lower bound on this worst case. Naturally, it remains an NP-hard problem to elicit all the disallowed assignments, but we at least have a theoretical upper bound on the size of the smallest graph that captures *all* the symmetries of the problem.

4.2 Minimising the number of literal nodes

CSPs can be simplified by using standard propagation techniques which reduce the domains of the variables and, thus, the number of literal nodes which appear in the full assignment graph. Correct simplifications to achieve node- and arc-consistency are well-known, and yield a new CSP that has the same set of solutions as the original one. Indeed, consistency algorithms were first devised for improving the efficiency of picture recognition programs, to reduce the size of the graph, which is exactly what we are seeking to do!

Perhaps surprisingly, these methods can also yield a loss of detected symmetries, i.e., they can exclude graph automorphisms which were present in the graph G of the original CSP but are eliminated from the graph G' of the simplified CSP. In particular, graph G may have an automorphism that maps a literal *lit* onto another literal $f(lit)$, while G' has node *lit* but not $f(lit)$.

Example 13 Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x \geq y, x \neq y, z > y\})$. While arc-consistency will eliminate value 1 from D_z and 3 from D_y (due to constraint $z > y$), the domain of x will remain unchanged after achieving arc-consistency (since all its values are supported), obtaining the arc-consistent CSP $(\{x, y, z\}, \{\{1, 2, 3\}, \{1, 2\}, \{2, 3\}\}, \{x \geq y, x \neq y, z > y\})$. If we now choose to represent each constraint of the original CSP by its disallowed assignments, the graph has an automorphism corresponding to the variable symmetry $\langle x, z\rangle \leftrightarrow \langle z, x\rangle$. However, the graph associated with the arc-consistent CSP no longer has this (or any) variable symmetry.

We have already motivated the need to merge all constraints with the same scope before generating the graph associated with a CSP. Such a merging for the previous example would have ensured that value 1 was also removed from D_x, thus preserving the variable symmetry between x and z. If we assume this preprocessing step has been carried out, we can show that any algorithm which achieves arc-consistency preserves all the variable and value symmetries that were present in the original CSP.

We now consider n-ary arc-consistency. A CSP is n-ary arc-consistent if every value d_i of every variable x_i is supported in every constraint c with x_i in its scope, because there is a tuple of values, each from the domain of its variable, which is allowed by c and assigns d_i to x_i.

Lemma 5 *Let f be an automorphism of the full assignment graph for a CSP (X, D, C) whose constraints all have distinct scopes. If f represents a variable or a value symmetry, then the graph of the n-ary arc-consistent version of the CSP has an automorphism representing the same symmetry as f.*

Proof Let $lit(x_i)$ be the set of literals associated with a variable $x_i \in X$. By assumption, f is a variable or value symmetry and, therefore, $\forall x_i \in X, \exists x_j \in X$ such that $lit(x_j) = f(lit(x_i))$ and $lit(x_i) = f(lit(x_j))$. Also, $\forall c \in C$, each assignment A over $vars(c)$ has an image assignment $f(A)$ over the variables $\{f(x) : x \in vars(c)\}$, which we will write $f(vars(c))$. Since each assignment in c has an image over the same set of variables $f(vars(c))$, and since each constraint has a different scope, we can call $f(c)$ the unique constraint over $f(vars(c))$. This constraint has the same number of tuples as c. Also, if c is an allowed constraint then so is $f(c)$, and if c is a disallowed constraint then so is $f(c)$. Note, finally, that since f is a one-to-one mapping of constraints, each constraint $c \in C$ is the image of another constraint $c' \in C$, i.e., $c = f(c')$.

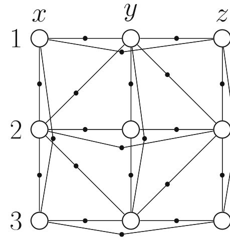
The proof will show that if f is a graph automorphism satisfying this condition, then a literal lit will be unsupported if and only if its image $f(lit)$ under the automorphism is also unsupported, and it will be supported by constraint c if and only if its image $f(lit)$ is also supported by $f(c)$. This shows that every such automorphism is preserved after establishing arc-consistency on the CSP. Let us first show that lit will be supported by constraint c if and only if its image $f(lit)$ is also supported by $f(c)$. The property clearly holds if c is an allowed constraint, by definition of automorphism. If c is a disallowed constraint, then lit is supported if there is an assignment A over $vars(c)$, whose literals belong to the current domains of their variables. Suppose the literals in $f(A)$ were linked to a disallowed assignment node, then A would be a disallowed assignment, which is false. Therefore, the literal nodes in $f(A)$ are not linked to a disallowed assignment node, and so $f(A)$ provides support for $f(lit)$ with respect to $f(c)$. In the other direction, if A' provides support for $f(lit)$ with respect to $f(c)$, then there exists $f(A) = A'$ and, by the same proof, A provides support for lit with respect to c.

To complete the proof, if lit is unsupported, then it has no support with respect to some constraint c and, therefore, $f(lit)$ has no support with respect to $f(c)$. If $f(lit)$ is unsupported, it has no support with respect to some constraint c'; $c' = f(c)$ for some constraint c; and lit has no support with respect to c. □

Having established that n-ary arc-consistency preserves variable and value symmetry, we show that there are other literal symmetries that are not preserved after establishing arc-consistency.

Example 14 Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, C)$, where $C = \{x \neq y, y \neq z, x \neq z, con(x, y), con(z, y)\}$, and con is defined by the following disallowed assignments: $\{\langle 2, 1\rangle, \langle 2, 3\rangle\}$.

Fig. 17 CSP with rotational symmetry



The disallowed assignments graph, illustrated in Fig. 17, admits the rotational symmetry: $\langle x=1, x=2, x=3, y=1, y=2, y=3, z=1, z=2, z=3\rangle \leftrightarrow \langle z=1, y=1, x=1, z=2, y=2, x=2, z=3, y=3, x=3\rangle$. The literal node $x=2$ can be removed because it is incompatible with every value of the variable y. However, the node $y=1$ cannot be removed because it is compatible with $x=3$ and with $z=3$. Our pruning procedure only removes two literal nodes $x=2$ and $z=2$ from the graph, and the disallowed assignments that contain them. As a result, the pruned graph no longer has the rotational symmetry exhibited by the original graph.

4.3 Combining both techniques to prune the graph

We briefly describe the method to reduce the number of nodes in the full assignment graph. The method can logically be divided into three steps:

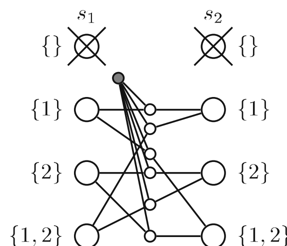
1. Rewrite the input CSP eliminating expressions and structured objects
2. Establish arc-consistency
3. Map the resulting CSP into a full assignment graph

In practise, the third and first steps are integrated, while arc-consistency is established within the graphical representation. As indicated in Section 3.1, constraints involving expressions are eliminated by simply flattening them into the required set of variable assignments.

The effect of pruning can be dramatic when eliminating literals of set variables that were represented using the extensional representation. This is indeed the case for cardinality constraints of the form $|x_i| = I$, where I is an integer constant, since assignment nodes can then only be created for literals $x_i = d_i$ for which $|d_i| = I$.

Example 15 Consider the CSP (X, D, C) where $X = \{s_1, s_2\}$, $D_{s_1} = D_{s_2} = \{\{\}, \{1\}, \{2\}, \{1,2\}\}$ (that is, $s_1, s_2 \subseteq \{1,2\}$) and $C = |s_1 \cap s_2| = 1$. The graph of this CSP using

Fig. 18 Pruning unnecessary values



the extensional representation is shown in Fig. 18. As can be seen in the figure, none of the assignments that satisfy the constraint involves $s_1 = \{\}$ or $s_2 = \{\}$. The literal nodes associated with these literals can thus be removed from the graph.

From a theoretical point of view, it is advantageous for the CSP on which the arc-consistency algorithm is applied to include global constraints with many variables in their scope. This is because establishing arc-consistency on an *all_different* constraint is more powerful—and prunes more domain values—than on the set of binary disequalities that are logically equivalent to it. From a practical standpoint, however, there are very few implementations of propagation on complex global constraints, such as the *cumulative*, or *cycle* constraint, that establish arc-consistency. Consequently, there is no guarantee for the properties of preserving variable or value symmetries to be preserved by current implementations of global constraints.

More work will be needed to establish a real understanding of the trade-offs between the extra pruning due to global constraint propagation, and any loss of symmetries that may result. What seems clear is that once the arc-consistency has been established, constraints should be rewritten and expressed using a logically equivalent representation with minimal constraint scopes. An automated system to perform this rewriting is future work.

5 Experimental evaluation

5.1 Implementation

We have implemented an automatic symmetry detection system for the subset of ECL^{i}PS^e programs [1] that only use finite domain and/or set constraints. The main components of this system are depicted in Fig. 19, with ovals representing input/output files, white rectangles representing system components, and shaded rectangles indicating external components used by the system.

The first component is an ECL^{i}PS^e library that receives as input the ECL^{i}PS^e program (possibly divided into the model file and the data file) and outputs a text file containing the set of (syntactic) constraints that would be posted to the solver during the execution of the program.

This file is, in turn, processed by a graph generator that obtains, for each constraint in the text file, three possible graph representations: the full assignments graph, a version of Puget's extensional graph, and a version of Puget's Boolean graph. This is done as follows. For the full assignments graph, equality constraints are represented by their allowed assignments, disequality constraints by their disallowed assignments, the *all_different* constraint is split into the equivalent conjunction of binary disequalities, sets are represented using the extensional representation, and cardinality constraints are represented using their allowed assignments. No other kinds of constraints are needed to represent all our benchmarks. For Puget's extensional representation, equality, disequality, sets and cardinality constraints are represented as before, the *all_different* constraint is represented using its disallowed assignments, expressions of the form $op(x)$ where x occurs only once are treated specially, and any other expression is treated using temporary variables as described in Section 2.4. Finally, for Puget's Boolean representation, each benchmark is converted into a

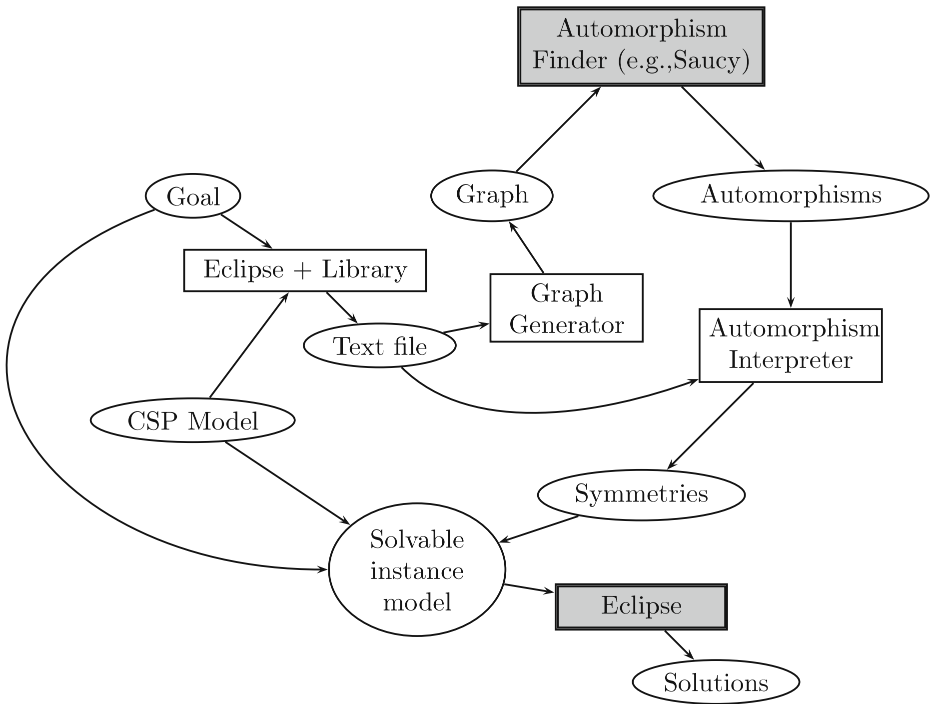


Fig. 19 System design

Boolean representation as indicated by Section 2.3. Constraint nodes for the original constraints and variable nodes for the original variables are not created.

Note that while generating the text file significantly slows down the process, it allows us to easily explore different alternatives for constructing the graph.

The resulting graph is input to the graph automorphism package Saucy [4] which returns the generating set of the automorphism group. One minor point must be considered: graph automorphism packages consider a graph to be labelled by non-negative integers, whereas our graph nodes have more descriptive labels. Therefore, our system creates a map from graph labels (e.g. $x = 2$) onto integers. This map is also used to convert the numeric labels of the automorphisms found back into descriptive graph labels, thus representing the symmetries in a more intuitive form.

Example 16 Consider the literal nodes of the graph shown in Fig. 20 for the four-queens problem. The label $(x_i = d_i)$ of each literal node is mapped to the positive integer shown within each node (for simplicity, the mapping for assignment nodes is omitted). For this graph, the output of Saucy (omitting the assignment nodes) is:

```
(1 4)(2 8)(3 12)(6 9)(7 13)(11 14)
(0 3)(1 2)(4 7)(5 6)(8 11)(9 10)(12 15)(13 14)
```

Fig. 20 Literal nodes of four-queens

	Q_1	Q_2	Q_3	Q_4
1	12	8	4	0
2	13	9	5	1
3	14	10	6	2
4	15	11	7	3

Each line represents one symmetry and each pair of numbers represents a swap of nodes. The first line corresponds to the diagonal variable-value symmetry: $\langle Q4 = 2, Q4 = 3, Q4 = 4, Q3 = 3, Q3 = 4, Q2 = 4, Q3 = 1, Q2 = 1, Q1 = 1, Q2 = 2, Q1 = 2, Q1 = 3\rangle \leftrightarrow \langle Q3 = 1, Q2 = 1, Q1 = 1, Q2 = 2, Q1 = 2, Q1 = 3, Q4 = 2, Q4 = 3, Q4 = 4, Q3 = 3, Q3 = 4, Q2 = 4\rangle$. The second line corresponds to the value symmetry $\langle 1, 2\rangle \leftrightarrow \langle 4, 3\rangle$, for each queen. These two generators can be composed to form the group that represents the eight symmetries of a square.

Although we conducted most of our experiments using Saucy to find graph automorphisms, our implementation is not tied to any particular package. Any graph automorphism package could be used in its place, such as Nauty [10] or AUTOM [13]. For instance, we have successfully tested Nauty with our implementation. We would have liked to have used the faster AUTOM [13], but it is not publicly available.

Section 2.3 described two possible ways of representing set variables: using an extensional and a Boolean representation. We use the former when evaluating the extensional meaning of constraints and when pruning, and the latter when producing a graph to be searched for automorphisms. This is because the latter yields automorphisms which reflect permutations of the possible elements rather than of the possible sets themselves, a form more suitable to be used as input to symmetry breaking packages such as GAP-SBDS [7].

Once the symmetries of a CSP have been found, they can be used to aid a search for the CSP's solutions. While the symmetries detected by our system are only of interest to us as stepping stones towards model-based symmetries, we wanted to connect our current system to ECL^{i}PS^e and GAP-SBDS to make sure everything was working properly. The automatic coupling of symmetry detection and symmetry breaking is complicated by the distinction between model and instance. Since the symmetries detected by our system are not applicable to the model, the system creates a new program composed of the original model, the goal that specifies the parameter values used as data for this instance, the symmetries that apply to the instance, and the search predicate that will be used to find a solution. This program can then be executed in ECL^{i}PS^e to solve the CSP instance. Only unique solutions—those that are not symmetrically equivalent to other solutions—are found.

5.2 Benchmarks

Let us now provide a brief summary of the set of benchmarks used in our experimental evaluation. In doing this we will follow the descriptions given in CSPLib [8].

Balanced incomplete block design A balanced incomplete block design is an arrangement of v distinct objects into b blocks such that each block contains exactly k distinct objects, each object occurs in exactly r different blocks, and every two distinct objects occur together in exactly λ blocks. Therefore, a BIBD is specified by five parameters, (v, b, k, r, λ). This benchmark is listed in the results as "bibd-v-b-k-r-λ".

We model this problem as a $v \times b$ binary matrix, with constraints that force exactly r ones per row, k ones per column, and a scalar product of λ between any pair of distinct rows. The symmetries found by all three implemented methods are:

- all blocks are interchangeable (variable symmetry)
- all objects are interchangeable (variable symmetry)

These correspond to permutations of the rows and columns of the binary matrix.

Social golfers The social golfers problem aims at scheduling g groups, with p golfers per group, over w weeks, in such a way that no golfer plays in the same group as any other golfer twice. This benchmark is listed in the results as "golf-w-g-p".

We model this problem using one set variable for each group, constraining each group to have cardinality p, and each intersection between any pair of distinct groups (from any weeks) to have cardinality at most one. The symmetries found by all three implemented methods are:

- all golfers are interchangeable (value symmetry)
- all weeks are interchangeable (variable symmetry)
- all groups within a single week are interchangeable (variable symmetry)

Golomb ruler A Golomb ruler is a set of m integers (marks on the ruler) $0 = a_1 < a_2 < ... < a_m$ such that the $\frac{m(m-1)}{2}$ differences $a_j - a_i$, $1 \leq i < j \leq m$ are distinct. One problem involving such rulers is to find a valid set of m marks. This benchmark is listed in the results as "golomb-m".

We model this problem using m integer variables and one integer variable per pairwise difference. The difference variables must be all different. A single symmetry is found using both implemented symmetry detection methods, corresponding to a 180° reflection of the ruler. This is a variable symmetry on the difference variables, and a variable-value symmetry on the marks variables.

***N*-queens** The N-queens problem is to place N queens on an $N \times N$ chessboard such that no queen attacks another. We model this problem using one integer variable per row in the board. Each value, from 1 in N, represents the column position of the queen in that row. This benchmark is listed in the results as "queens-N".

Puget's Boolean method and our method find all symmetries of a square. Puget's extensional method finds only the variable symmetry and value symmetry, and misses the non-compositional variable-value symmetry. In terms of the chessboard, it finds the horizontal and vertical reflections but not the rotational symmetry.

Latin square A Latin square is an $n \times n$ matrix where each element is a value from 1 to n. Each value must occur exactly once in each column and exactly once in each row. The problem is to find such a square for a given n. This benchmark is listed in the results as "latin-n".

We model this problem as an $n \times n$ matrix of integer variables with domain 1 to n. An *all_different* constraint is posted on each row and each column. The symmetries found by Puget's Boolean method and our method are:

- all rows are interchangeable (variable symmetry)
- all columns are interchangeable (variable symmetry)
- all values are interchangeable (value symmetry)
- the row and column dimensions are transposable (variable symmetry)
- the row and value dimensions are transposable (variable-value symmetry)

As for N-queens, Puget's extensional method finds the variable and value symmetries, but not the non-compositional variable-value symmetry.

Most perfect magic square A most perfect magic square is an arrangement of n^2 integers, 1 to n^2, into an $n \times n$ matrix such that the n numbers in all rows, columns and diagonals (with wrap-around) have the same sum, each 2 by 2 subsquare (with wrap-around) sums to $2(n^2 - 1)$, and all pairs of numbers distant $\frac{n}{2}$ on a diagonal sum to $n^2 - 1$. The problem aims at finding such a square for a given n. This benchmark is listed in the results as "mostperfect-n".

We model this problem as an $n \times n$ matrix of integer variables with domain 1 to n^2. Sum constraints are posted on the rows, columns and diagonals to enforce the magic-square property. Additional sum constraints over all 2 by 2 subsquares, and on the pairs of numbers on the major diagonals, enforce the most-perfect property.

This model resulted in a graph for $n = 4$ that was too large for our implementations of either of Puget's methods. Using our method, the symmetries found are:

- the symmetries of a square (rotations through 90, 180 and 270 degrees and reflections about the horizontal and vertical axes) (variable symmetry)
- the rows (or columns) can be cycled (variable symmetry)
- value i is interchangeable with value $n^2 - i - 1$ (value symmetry)

Steiner triples The Steiner triple problem of order n consists of finding a set of $\frac{n(n-1)}{6}$ triples of distinct integers from 1 to n, such that any pair of triples has at most one element in common. This benchmark is listed in the results as "steiner-n". The symmetries found by all three implemented methods are:

- all triples are interchangeable (variable symmetry)
- all values are interchangeable (value symmetry)

$N \times N$-queens The $N \times N$-queens problem is to place a coloured queen on every square of an $N \times N$ chessboard so that no two queens of the same colour attack each other. There are N colours. A solution to this problem is equivalent to N simultaneous non-overlapping solutions to the N-queens problem. This benchmark is listed in the results as "nnqueens-n". The symmetries found by all three implemented symmetry detection methods are:

- the symmetries of the chessboard (variable symmetry)
- all colours are interchangeable (value symmetry)

Graceful graph The graceful graph problem is to find a labelling f of the vertices of a graph such that f assigns each vertex a unique label from $\{0, 1, \ldots, e\}$ (where e is

the number of edges in the graph), and with each edge (a, b) labelled by $|f(a) - f(b)|$, all the edges labels are different. This benchmark is listed in the results as "graceful-m-n" for the graph $K_m \times P_n$. The symmetries found by all three implemented methods are:

- the symmetries of the graph itself (variable symmetry)
- the value symmetry that swaps a with $x - a$, where x depends on the particular instance (value symmetry)

5.3 Results for symmetry detection

Tables 1 and 2 show the results of our experimental evaluation of the automatic symmetry detection tool. Each row in the tables corresponds to a different instance of a benchmark problem described in the previous section. A bold font indicates the best result for that row in the table.

The columns in Table 1 compare the total number of nodes (Nodes) and the total number of edges in the graph (Edges) when using our implementations of Puget's

Table 2 Running times

Instance	Running time								
	Puget's (Ext)			Puget's (Bool)			Ours		
	Total	Gr	HR	Total	Gr	HR	Total	Gr	HR
bibd-3-3-1-1-0	0.04	0.50	0.50	0.04	0.50	0.50	**0.02**	.50	.50
bibd-6-10-5-3-2	20.80	0.90	0.07	20.61	0.90	0.07	**1.96**	.83	.14
golf-2-2-2	0.50	0.78	0.20	0.50	0.78	0.20	**0.18**	.72	.28
golf-2-2-3	17.16	0.89	0.09	16.90	0.89	0.08	**2.71**	.73	.23
golf-2-3-2	44.68	0.87	0.10	44.14	0.87	0.10	**6.72**	.72	.22
golf-3-2-2	1.36	0.76	0.21	1.32	0.76	0.21	**0.56**	.71	.25
golomb-4	0.28	0.86	0.14	**0.26**	0.85	0.15	0.41	.85	.12
golomb-5	1.41	0.91	0.08	**1.31**	0.90	0.08	2.00	.91	.08
golomb-6	5.36	0.93	0.05	**5.03**	0.94	0.05	7.67	.93	.05
golomb-7	16.97	0.95	0.04	**15.90**	0.95	0.03	24.45	.94	.03
graceful-3-2	0.22	0.68	0.27	**0.19**	0.68	0.32	0.31	.71	.26
graceful-5-2	6.20	0.76	0.19	**4.95**	0.82	0.15	8.41	.82	.14
latin-10	1.68	0.34	0.51	**0.46**	0.26	0.70	2.78	.48	.41
latin-11	2.53	0.33	0.49	**0.61**	0.25	0.70	4.27	.47	.40
latin-12	3.73	0.33	0.47	**0.78**	0.24	0.72	6.37	.47	.39
latin-13	5.71	0.30	0.42	**1.01**	0.25	0.70	9.17	.46	.37
latin-14	7.50	0.31	0.43	**1.31**	0.25	0.70	12.86	.46	.36
mostperfect-4	–	–	–	–	–	–	**31.70**	.85	.10
nnqueens-4	0.03	0.66	0.33	**0.01**	0.00	1.00	0.05	.60	.40
nnqueens-5	0.09	0.44	0.55	**0.05**	0.60	0.40	0.12	.58	.42
nnqueens-6	0.20	0.50	0.50	**0.08**	0.38	0.62	0.30	.60	.33
queens-10	0.08	0.63	0.38	**0.03**	0.33	0.67	0.15	.73	.27
queens-20	0.74	0.62	0.30	**0.13**	0.31	0.69	1.61	.80	.16
queens-30	2.74	0.63	0.25	**0.33**	0.33	0.67	6.62	.82	.13
queens-40	7.04	0.63	0.22	**0.65**	0.35	0.63	18.43	.84	.11
steiner-5	1.42	0.86	0.12	1.39	0.86	0.12	**0.46**	.74	.24
steiner-6	28.12	0.88	0.09	27.89	0.88	0.09	**5.92**	.74	.21
steiner-7	488.38	0.93	0.05	492.85	0.93	0.05	**57.49**	.76	.17

extensional method (Puget's (Ext)), Puget's Boolean method (Puget's (Bool)) and our method (Ours). Also, but only for our method, it shows the number of generators found for each instance (Gens).

The columns in Table 2 show the total running time in seconds (Total), followed by a breakdown (expressed as a proportion of the total time) indicating where this time is spent. In particular, the table shows the proportion of time spent in graph generation including the computation of the extensional constraints plus the time spent printing the graph to be input to Saucy (Gr) and the proportion of time taken to read Saucy's output information and print our human-readable form (HR). Any time not accounted for by these two columns is spent running Saucy, and is usually small in comparison. Again, there are three sets of data; one for Puget's extensional method (Puget's (Ext)), one for Puget's Boolean method (Puget's (Bool)) and one for our method (Ours). Running times were measured on a desktop with a 3GHz Intel Pentium 4 CPU and 2 GB RAM, running Linux kernel 2.4.22.

The results show that Puget's Boolean method is much more efficient when the problem has all-different constraints (e.g. the Latin square instances), since it handles these constraints specially. When the problem has many complex expressions (e.g. the Social Golfers instances), our method is more efficient because it avoids having many temporary variables.

We were unable to run the most perfect magic square problem using either of Puget's methods. The model we used has constraints of the form $x_1 + x_2 + x_3 + x_4 = c$ where the x_i are variables and c is a constant. For the size 4 instance, each variable has a domain of 16 values. To represent the addition as a temporary variable, each assignment over $\{x_1, x_2, x_3, x_4\}$ is represented in extension, resulting in $16^4 = 65536$ combinations with one node and five edges per combination. As there are several of these constraints, the resulting graph is too large for our implementation to handle.

However, Puget reports that his method can handle this problem very efficiently. There are several possible explanations for this discrepancy: (a) he may use a special representation of these "sum" constraints, like for all-different, (b) he may detect only variable symmetries in his experiments, or (c) his implementation may be more efficient than ours (for example, using AUTOM instead of Saucy). While (a) is the most likely answer, we do not see any natural way to model $x_1 + x_2 + x_3 + x_4 = c$ as a conjunction of Boolean constraints. Thus, Puget's second Boolean model - without variable nodes or constraint nodes - is not naturally applicable. It is also possible that Puget used a combination of the standard Boolean method for this constraint, and the conjunctive Boolean model for the *all_different* constraint, but Puget offers no proof that for a graph combining both kinds of Boolean representation, its automorphisms correspond to symmetries of the CSP. Indeed, the two Boolean representations associate a different semantics to a literal node, so having both in the same graph seems problematic.

5.4 Results for symmetry breaking

As mentioned before, we used the symmetries detected by our implementation to automatically break symmetries during search, using the GAP-SBDS [7] library for ECL^{i}PS^e. Results of the experiments for symmetry breaking are shown in Table 3. For each benchmark, times are shown for finding all solutions without and with SBDS. The numbers in parentheses show the ratio obtained by dividing the time with

Table 3 Running times to find all solutions with and without SBDS

Instance	Running time (seconds)			Detect
	No SBDS	SBDS (ratio)		(seconds)
nnqueens-7	>30 min	**1.62**	(–)	0.7
steiner-7	392.55	**0.87**	(0.002)	58.0
bibd-7-7-3-3-1	157.89	**1.01**	(0.006)	0.55
graceful-4-2	296.02	**2.57**	(0.008)	2.2
golf-3-3-2	76.34	**0.77**	(0.01)	21.0
queens-13	**48.23**	118.09	(2.45)	0.5
golomb-6	**8.73**	67.89	(7.78)	7.5
latin-8	**81.12**	647.21	(7.98)	1.0

SBDS by the time without SBDS. Note that the times do not include the time to find the symmetries, although this detection time is shown in a separate column. We have separated these times because the techniques developed in this paper are targeted towards finding symmetries in *models* - though in this paper we only find symmetries in problem instances. If we can achieve this aim, the time to find the symmetries will be amortised over all the problem instances in the class defined by the model, and in this case will be small in comparison with problem-solving time.

The aim of this section is not to demonstrate that speedups can be achieved by symmetry breaking methods (this has been the subject of many other papers, e.g. [5, 14, 15]) but, rather, to show that our system is implemented and that the output of our symmetry detection tool can be easily integrated with a symmetry breaking tool, such as GAP-SBDS. Still, as shown in Table 3, SBDS performs much better than a simple search when finding all solutions for more than half the benchmarks, with most speedups being of several orders of magnitude. For the rest, the overhead of symmetry breaking is greater than the time saved by reducing the search space.

6 Conclusion

Symmetry as graph automorphisms This paper has explored the extension and application of constraint symmetry detection based on graph automorphisms, and its integration into an algorithm that exploits these symmetries during search.

Previous symmetry detection using graph automorphisms can be divided into two main directions. The first dating back to Crawford et al. [3] represents constraints as sets of disallowed tuples. This approach is simple but can result in large graphs since a simple constraint (such as an equation) may require a great many disallowed edges in its representation. Consequently, this paper has explored the use of allowed edges, as well, for representing constraints. In doing so, it has studied a second approach, recently espoused in a sophisticated form by Puget [13], that represents constraints more flexibly. This flexibility can be exploited to keep the graph small, but it could lead to either representing too few symmetries, or - worse still - too many.

A flexible, powerful, and correct graph representation Too few symmetries may be represented if the graph includes a node for each variable, so that only (combinations of) value and variable symmetries can be represented. Therefore, this paper has described a graph representation without such nodes. The drawback of such a representation is that a graph automorphism could map sets of nodes representing

solutions to the original problem, to sets of nodes which do not represent a solution, because they do not "cover" all the variables. Accordingly, we imposed sufficient conditions on the new graph representation to preclude such automorphisms.

Too few symmetries may also be represented if the graph distinguishes different (kinds of) constraints. This would prevent, for example, a disequation from being involved in a symmetry with an *all_different* constraint. To maximise the number of potential symmetries, the graph representation introduced in this paper made no distinction between different constraints. Moreover, it made no distinction between an edge connecting two literals explicitly allowed by a binary constraint, or allowed because their associated variables do not belong to the scope of any constraint. Furthermore, it made no distinction between an edge connecting two literals explicitly disallowed due to a constraint, or disallowed because they represent distinct values for the same variables. The main drawback of such a representation is the sheer size of the resulting graph: in principle an allowed edge is required for every compatible set of nodes, and a disallowed edge between every incompatible set. Therefore, this paper introduced a graph representation that uses as few edges as possible while still maximising the number of potential symmetries. Since this requires all constraints to be represented extensionally, keeping the size of the graph as small as possible is very important, particularly for variables with large domains, such as set variables.

Too many symmetries are represented by a graph if it has automorphisms that do not correspond to symmetries of the CSP. This can arise, for example, if the existence of an edge does not have a unique meaning. For example, the existence of two edges representing the only two allowed tuples for a single constraint mean that *one* should be in a solution, while the existence of two edges representing the only allowed tuples from two different constraints mean that *both* should be in a solution. Also, the absence of an edge should have a unique meaning, such as that the two unconnected nodes are unconstrained, or that they are incompatible (e.g. if they represent two different values for a variable).

For a graph with edges representing allowed tuples of a constraint, but without variable nodes or constraint nodes, it is not trivial to avoid having too many symmetries. However, for the *full assignment* graph introduced in this paper, it was proven that every graph automorphism corresponds to a problem symmetry.

Reducing the size of the graph The paper introduced two additional ways of reducing the size of the graph representation. The first approach is to achieve arc-consistency on the original problem, and build a graph representing this reduced problem. The paper showed that achieving arc-consistency by reducing the domains of the variables preserves the variable and value symmetries detected by our approach. However, it also gave an example to show that non-compositional variable-value symmetry may be lost as a result of achieving arc-consistency.

The second approach is to represent a constraint by a logically equivalent conjunction of constraints, each with a smaller scope. This was shown to reduce the size of the graphical representation, and potentially improve pruning.

In summary, when compared to Puget's approach, the full assignment graph is more restricted than the combination of Puget's different approaches: extensional, intensional, standard Boolean, and conjunctive Boolean. Indeed, the full assignment graph admits only allowed and disallowed extensional constraints. Nevertheless, our approach can be applied to either Boolean model of the problem. The resulting

benefit is that we have been able to eliminate variable nodes and constraint nodes from the full assignment graph, so as to be able to capture non-composable variable-value symmetries and, at the same time, obtain proofs that graph automorphisms correspond to symmetries of the CSP even when combinations of allowed and disallowed constraints are represented. Puget never attempted such a proof and, without the restrictions introduced in this paper, it is shown that certain combinations would lead to graphs whose automorphisms did not correspond to symmetries of the CSP. Moreover, and perhaps surprisingly, it is shown that the extensional graph representation does not necessarily lead to larger graphs than the intensional representation for the Boolean model: even for the *all_different* constraint exampled by Puget, there is an extensional representation with comparable size.

Implementation and next objectives The whole system has been implemented using the CLP system ECL^{i}PS^e and the graph automorphism package Saucy. The CSP is automatically transformed into a graph, the automorphisms are elicited and expressed as constraint symmetries of the CSP, and the CSP is solved using the discovered symmetries to automatically prune the search tree. Experiments were performed on a range of benchmarks to establish the correctness of the implementation.

The approach presented in this paper is designed to be able to detect as wide a class of constraint symmetries as possible. The efficiency of the symmetry detection method has been a secondary consideration, since it is not planned to be used to detect and apply symmetries for each new problem instance. Rather, the aim is to detect as many representational symmetries as possible for several problem instances, so that they can be tested against a generic problem model, to determine which ones hold for the whole problem class. These symmetries can then be used to accelerate the solving of any instance of the model. Accordingly, the cost of detecting symmetries can be amortised across all the instances of the problem which are eventually solved using the detected symmetries.

Further, the resulting model-based approach will be able to scale up the applicability of our automatic symmetry detection system, since the detected symmetries can be used to accelerate the solving of large practical problems involving hundreds or thousands of variables and constraints.

References

1. Apt, K. R., & Wallace, M. G. (2006). *Constraint logic programming using ECLiPSe*. Cambridge University Press.
2. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K. E., & Smith, B. M. (2005). Symmetry definitions for constraint satisfaction problems. In P. van Beek (Ed.), *CP2005, LNCS* (Vol. 3709, pp. 17–31). Springer.
3. Crawford, J., Ginsberg, M. L., Luck, E., & Roy, A. (1996). Symmetry-breaking predicates for search problems. In L. C. Aiello, J. Doyle, S. Shapiro (Eds.), *KR'96: Principles of knowledge representation and reasoning* (pp. 148–159). San Francisco, CA: Morgan Kaufmann.
4. Darga, P. T., Liffiton, M. H., Sakallah, K. A., & Markov, I. L. (2004). Exploiting structure in symmetry detection for CNF. In S. Malik, L. Fix, A. B. Kahng (Eds.), *DAC* (pp. 530–534). ACM.
5. Freuder, E. C. (1991). Eliminating interchangeable values in constraint satisfaction problems. In *Proc. AAAI'91* (Vol. 1, pp. 227–233).
6. Frisch, A. M., Miguel, I., & Walsh, T. (2003). CGRASS: A system for transforming constraint satisfaction problems. In B. O'Sullivan (Ed.), *Recent advances in constraints, joint ERCIM/CologNet*

international workshop on constraint solving and constraint logic programming, LNCS (Vol. 2627, pp. 15–30).
7. Gent, I. P., Harvey, W., & Kelsey, T. (2002). Groups and constraints: Symmetry breaking during search. In P. van Hentenryck (Ed.), *CP2002, LNCS* (Vol. 2470, pp. 415–430). Springer.
8. Gent, I. P., & Walsh, T. (1999). CSPLib: A benchmark library for constraints. Technical report, Technical report APES-09-1999. A shorter version appears in the proceedings of the 5th international conference on principles and practices of constraint programming (CP-99). Available from http://www.csplib.org/.
9. Jégou, P. (1993). Decomposition of domains based on the micro-structure of finite constraint satisfaction problems. In *AAAI93: Proceedings of the 11th national conference on artificial intelligence* (pp. 731–736).
10. McKay, B. D. (1981). Practical graph isomorphism. *Congressus Numerantium*, *30*, 45–87.
11. Pearson, J., van Hentenryck, P., Flener, P., & Ågren, M. (2005). Compositional derivation of symmetries for constraint satisfaction. In *Proceedings of the international symposium on abstraction, reformulation, and approximation (SARA'05)*.
12. Puget, J.-F. (2002). Symmetry breaking revisited. In P. Van Hentenryck (Ed.), *CP2002, LNCS* (Vol. 2470, pp. 446–461). Springer.
13. Puget, J.-F. (2005). Automatic detection of variable and value symmetries. P. van Beek (Ed.), *CP2005, LNCS* (Vol. 3709, pp. 475–489). Springer.
14. Puget, J.-F. (2005). Breaking all value symmetries in surjection problems. In P. van Beek (Ed.), *CP2005, LNCS* (Vol. 3709, pp. 490–504). Springer.
15. Ramani, A., & Markov, I. L. (2004). Automatically exploiting symmetries in constraint programming. In B. Faltings, A. Petcu, F. Fages, F. Rossi (Eds.), *CSCLP* (Vol. 3419, pp. 98–112).
16. Roy, P., & Pachet, F. (1998). Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *ECAI98 workshop on non-binary constraints* (pp. 27–33).