Lightweight Detection of Variable Symmetries for Constraint Satisfaction

Christophe Lecoutre, Sébastien Tabary

Lightweight Detection of Variable Symmetries for Constraint Satisfaction

Christophe Lecoutre Sebastien Tabary
CRIL – CNRS UMR 8188
Université Lille-Nord de France, Artois
rue de l'université, SP 16, F-62307 Lens, France
{$lecoutre, tabary$}@cril.fr

Abstract

In this paper, we propose to automatically detect variable symmetries of CSP instances by computing for each constraint scope a partition exhibiting locally symmetric variables. From this local information obtained in polynomial time, we can build a so-called lsv-graph whose automorphisms correspond to (global) variable symmetries. Interestingly enough, our approach allows us to disregard the representation (extension, intension, global) of constraints. Besides, the size of the lsv-graph is linear with respect to the number of constraints (and their arity).

1. Introduction

Symmetry breaking [3] is an important research topic in Constraint Programming. By discarding symmetric parts of a constraint network, one may obtain a dramatic reduction of the search effort required to find a solution or to prove unsatisfiability. Different approaches have been proposed to exploit symmetries: one can a) add symmetry breaking constraints before search [4], b) break symmetries during search (SBDS) [2, 10], c) break symmetries via dominance detection (SBDD) [6, 7]. However, in this line of research, symmetries are considered to be given by the user. Consequently, to build black-box solvers (i.e. solvers that can be easily handled by non-experts), one important issue has to be addressed: automatically discovering symmetries.

In [9], the system CGRASS is proposed to analyze CSP instances in order to automatically identify symmetries and implied constraints. At the heart of the system, a syntactic comparison is performed using the computation of *normal forms*. To detect symmetric variables, each pair of variables is considered, and the set of constraints obtained after swapping them is normalized and compared with the original form. Unfortunately, this approach is impractical in general [9], due to the complexity of computing normal forms over all pairs of variables and over the full set of constraints.

The automatic detection of symmetries has also been addressed using software computing graph automorphisms. In [15], the authors propose to reduce CSP instances into SAT ones while capturing their symmetrical structure (before reduction). A parse graph is built from the predicate expressions associated with the constraint set, and some transformations (e.g. removing brackets, grouping operators) are proposed to favor the detection of symmetries. The automorphisms of this graph are then computed using a program such as Saucy [5]. In this vein, an automatic symmetry detection method has been proposed in [14]: it allows us to detect value and variable symmetries and non trivial ones involving both variables and values.

In this paper, we propose to automatically detect variable symmetries by computing a partition of the scope of each constraint exhibiting locally symmetric variables. From this local information computed in polynomial time, we build a so-called lsv-graph whose automorphisms correspond to (global) variable symmetries. Interestingly, our approach allows us to disregard the representation of constraints: whatever the representation is (extension, intension, global), one exploits the same notion (kind of nodes) in the lsv-graph. Besides, both the number of nodes and edges in the lsv-graph is linear with respect to the number of constraints (and their arity) of the CSP instance, making symmetry detection efficient when using available tools such as Nauty [13] and Saucy [5] on very large instances. Using generators of the symmetry group returned by the graph automorphism identification software, lexicographic ordering constraints can be posted.

2. Technical Background

A Constraint Network (CN) P is a pair $(\mathscr{X}, \mathscr{C})$ where $\mathscr{X}$ is a finite set of n variables and $\mathscr{C}$ a finite set of e constraints [12]. Each variable $X \in \mathscr{X}$ has an associated domain, denoted by $dom(X)$, that contains the set of values allowed for X. Each constraint $C \in \mathscr{C}$ involves an ordered subset of variables of $\mathscr{X}$ and has an associated relation, de-

noted by $rel(C)$, which is the set of tuples allowed for this subset of variables. This subset of variables is the *scope* of C and is denoted by $scp(C)$. The *arity* of a constraint is the number of variables in its scope. A *binary* constraint has arity 2. A *solution* to a CN is an assignment of a value to each variable such that all the constraints are satisfied.

Given an ordered set $\{X_1, \ldots, X_i, \ldots, X_k\}$ of k variables and a k-tuple $\tau = (a_1, \ldots, a_i, \ldots, a_k)$ of values, the individual value a_i will be denoted by $\tau[X_i]$. $\tau_{X_i \leftrightarrow X_j}$ denotes the tuple obtained from τ by swapping $\tau[X_i]$ with $\tau[X_j]$.

Even if mathematically it is useful to consider a constraint C defined by a relation, in practice it can be defined extensionally by a table (i.e. an explicit list of allowed tuples) or intensionally by a predicate expression denoted by $pre(C)$. The size of a table and an expression corresponds to the number of its elements and its tokens (operators, constants and variables), respectively.

In the domain of constraint satisfaction, many definitions (see [3]) of symmetry have been proposed. Roughly speaking, a symmetry is a bijection which, when applied, lets unchanged a given object (for example, a graph). In this paper, we restrict our attention to variable symmetries:

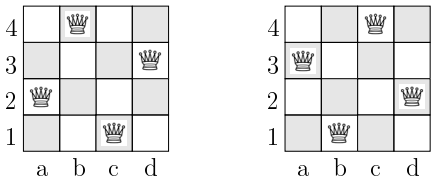
Definition 1 *Let $P = (\mathscr{X}, \mathscr{C})$ be a CN with $\mathscr{X} = \{X_1, \ldots, X_n\}$. A variable symmetry σ of P is a permutation on $\mathscr{X}$ such that $\{X_1 = a_1, \ldots, X_n = a_n\}$ is a solution of P iff $\{\sigma(X_1) = a_1, \ldots, \sigma(X_n) = a_n\}$ is a solution of P.*

The set of (variable) symmetries of a given CN forms a group. For example, the inverse of a symmetry, as well as the composition of two symmetries, are symmetries. A group can be represented by means of a subset of its elements called generators. Generators allow compact representations of sets of symmetries. Every symmetry can be expressed using a composition of these generators. Finally, every variable symmetry can be described by a set of cycles of the form $(X_{i_1}, X_{i_2}, \ldots, X_{i_k})$ which means that the variable X_{i_j} is mapped to $X_{i_{j+1}}$ for $j \in 1..k-1$, and the variable X_{i_k} is mapped to X_{i_1}. For example, $\sigma = \{(X_1, X_3), (X_2, X_4, X_5)\}$ contains two cycles and represents $\sigma(X_1) = X_3, \sigma(X_2) = X_4, \sigma(X_3) = X_1, \sigma(X_4) = X_5, \sigma(X_5) = X_2$.

Interchangeability is an important property [8] defined on values that corresponds to a particular form of (value) symmetry. However, this property can also be defined on variables (see e.g. [11]). A variable X is (fully) interchangeable with a variable Y (on a CN P) iff for every solution S of P, $S_{X \leftrightarrow Y}$ is also a solution of P where $S_{X \leftrightarrow Y}$ is obtained from S by swapping the values of the variables X and Y. Variable and value interchangeability is sometimes called pairwise or piecewise variable and value symmetry.

To break variable symmetries, lexicographic ordering constraints, which are defined on two vectors of variables, can be posted [4]. When variables correspond to letters, the two vectors represent words and we obtain the classical order used by dictionaries.

As an illustration, let us consider the $4\text{-}queens$ instance (modelled as a binary CN): we have to put four queens on a 4×4 board such that no two queens attack each other. There is one variable per queen (column) and the values are row numbers. Denoting the variables by X_a, X_b, X_c and X_d, to clarify the correspondence with columns, Figure 1 shows the two solutions for this instance. The first is $\{X_a = 2, X_b = 4, X_c = 1, X_d = 3\}$ and the second is $\{X_a = 3, X_b = 1, X_c = 4, X_d = 2\}$. If we disregard the identity permutation, only f^h is a variable symmetry defined in cyclic form by $f^h = \{(X_a, X_d), (X_b, X_c)\}$.



(a) First Solution (b) Second Solution

Figure 1. Solutions of the $4\text{-}queens$ instance.

3. Detecting Variable Symmetries

In order to automatically detect variable symmetries, we propose an original representation of any CN by a so-called lsv-graph (lsv stands for locally symmetric variables) which outlines many variable symmetries of the network. To summarize our approach, we proceed in three steps. The first one corresponds to a local analysis of each constraint in order to identify locally symmetric variables. The second one corresponds to the construction of a lsv-graph, and in the third one, generators for the symmetry group are computed using a graph automorphism identification algorithm, as proposed in [4, 15, 14].

3.1. Locally Symmetric Variables

Two variables involved in a constraint C are locally symmetric iff it is possible to permute both variables without modifying the set of allowed tuples.

Definition 2 *Two variables X and Y are locally symmetric for a constraint C iff $\{X, Y\} \subseteq scp(C)$ and $\forall \tau \in rel(C), \tau_{X_i \leftrightarrow X_j} \in rel(C)$.*

As symmetry is transitive, one can compute for each constraint a partition of its scope, each element of this partition being a set of pairwise symmetric variables. Function

Algorithm 1: computeSymmetricVariables(C: $Constraint$): Partition

```
1 Δ ← ∅ ; S ← scp(C)
2 while S ≠ ∅ do
3     pick X from S
4     T ← {X}
5     foreach Y ≠ X ∈ S do
6         if isLocallySymmetric(C, X, Y) then
              T ← T ∪ {Y}
7     S ← S \ T ; Δ ← Δ ∪ T
8 return Δ
```

$computeSymmetricVariables$ (Algorithm 1), identifies locally symmetric variables for any extensional, intensional or global[1] constraint C. A set S is first initialized with all variables involved in C. At each turn of the main loop, the algorithm picks a variable X from S, and computes the set T of variables symmetric with X. Next, T is subtracted from S and added to the partition Δ currently built.

At the heart of Algorithm 1, there is a call to function $isLocallySymmetric$ (Algorithm 2). For a given constraint C and two variables X and Y involved in C, it simply determines whether X and Y are locally symmetric for C or not. Generally speaking, three cases have to be considered depending on the representation of C:

1. If C is extensional (lines 1 to 4), the algorithm builds for each tuple in the table associated with C, a new one by swapping the values of the variables X and Y and then checks if it belongs to $rel(C)$.
2. If C is intensional (lines 5 to 7), the algorithm builds a canonical tree representation of the predicate expression $pre(C)$ associated with C by a call to the function $buildCanonicalTree$ (see Section 3.2). A second canonical tree representation is built after swapping variables X and Y in $pre(C)$, denoted by $pre(C)_{X \leftrightarrow Y}$. Both canonical representations are then compared, and X and Y are identified as being locally symmetric for C iff they are identical.
3. If C is a global constraint (line 8 to 11), a specific treatment must be performed. For example, any two variables involved in an "allDifferent" constraint are locally symmetric. Two variables involved in a "weightedSum" constraint are locally symmetric if they have the same attached coefficient.

A partition of the scope of a constraint (defined in extension or intension) can be computed in polynomial time. Indeed, we have the following complexity result:

[1]We illustrate our purpose with two global constraints, namely, allDifferent and weightedSum.

Algorithm 2: isLocallySymmetric(C: $Constraint$, X, Y: $Variable$): Boolean

```
1  if C is defined in extension then
2      foreach τ ∈ rel(C) do
3          if τ_{X↔Y} ∉ rel(C) then return false
4      return true
5  if C is defined in intension then
6      t ← buildCanonicalTree(pre(C))
7      return t = buildCanonicalTree(pre(C)_{X↔Y})
8  if C is a global constraint then
9      if C is AllDifferent then return true
10     if C is WeightedSum then return
       coefficient(X) = coefficient(Y)
11     ...
```

Theorem 1 *The worst-case time complexity of $computeSymmetricVariables$ for a constraint C of arity r is:*

- *$O(r^2 t\gamma)$ if C is extensional with t being the size of the table associated with C and γ the complexity of performing a constraint check ;*
- *$O(r^2 p^2 log(p))$ if C is intensional with p being the size of $pre(C)$.*

In practice, one may even expect a better behavior than the ones predicted in the worst case. First, the number of calls to $isLocallySymmetric$ is only $r - 1$ when the constraint is fully symmetric (i.e. all variables are locally symmetric), due to transitivity. Second, for an intensional constraint, $buildCanonicalTree$ is reduced to $O(p^2)$ when operators are binary. Finally, for an extensional constraint, when two variables are not symmetric, one may quickly exit the $foreach$ loop of Algorithm 2.

3.2. Computing Normal Forms of Predicates

As previously mentioned, normal forms have already been proposed [9] to identify symmetric variables of a CN. However, this approach was applied globally to the set of constraints, making it impractical in practice (except for small instances). Here, we propose to apply a similar approach on each constraint individually. It can so be applied quite efficiently, except for some very specific cases where predicate expressions or constraint arities are very large.

To make our presentation concrete, we consider here the grammar[2] used in CSP solver competitions to build predicate expressions. Many operators on which expressions are built are both commutative and associative: add (+), mul

[2]http://cpai.ucc.ie/08/XCSP2_1.pdf

(*), min, max, and, or, xor, iff, eq (=), ne ($\neq$). Here are some simple rewriting rules that we propose to apply:

- group associative operators using n-ary equivalent operators [15]. For example, replace $add(X, add(Y, Z))$ by $add(X, Y, Z)$.
- replace all occurrences of ge ($\geq$) and gt ($>$) by le ($\leq$) and lt ($<$) [9]. For example, replace $ge(X, Y)$ by $le(Y, X)$.
- replace the sequence 'abs sub' with a new commutative operator 'abssub' combining both operators. For example, replace $abs(sub(X, Y))$ by $abssub(X, Y)$.

Interestingly, it is possible to build a parse tree (each node is labelled with a token of the expression) in one pass while taking into account all rules indicated above. Even if some additional sophisticated rules may be imagined (e.g. one may adopt specific rules for linear and non-linear equations), we believe that this simple set of rules is sufficient to capture symmetric variables of many constraints. Also, notice the importance of the new operator 'abssub' as it occurs in various series of instances (e.g. frequency assignment problems). This new operator being commutative, we can identify symmetries undetected when the (non-commutative) operator sub is present.

To obtain a canonical form from an initial parse tree, it suffices to render canonical the root of the tree. A node is made canonical as follows: first, all children (if any) are made canonical, and sorted if the label associated with the node corresponds to a commutative operator. To obtain a normal form, it is necessary to define a total order over the set of operators, integers and variables. This order can be built rather naturally [9].

Except when the size of the predicate expression is very large, computing a tree in canonical form is cheap. Indeed, we have the following complexity result:

Theorem 2 *The worst-case time complexity of building a tree in canonical form, from a predicate expression $expr$, is $O(p^2 log(p))$ where p denotes the size of $expr$.*

3.3. Constructing lsv-graphs

Once locally symmetric variables have been identified for each constraint (through partitioning), one can build a colored graph dedicated to the search of variable symmetries for the given CN. Each automorphism in the graph corresponds to a variable symmetry in the network. This is an approach introduced in [4] and exploited, for example, in [1, 15, 14]. Typically, all automorphisms can be obtained by composition from a subset of automorphisms called generators. We show here that the colored graphs we build are of limited size while capturing many variable symmetries.

The construction of colored graphs, denoted by lsv-graphs from now, we propose is as follows. Each variable of the given CN P is represented with a node, denoted by "variable-node". For each constraint of arity r, we add a "constraint-node" and r "binding-nodes", one for each variable involved in the constraint. Binding-nodes allow to connect constraint-nodes with variable-nodes: if C is a constraint involving X then we have a connection between the constraint-node corresponding to C and the variable-node corresponding to X though a binding-node.

A color is associated with each node of the graph (permutations are only allowed between nodes of the same color). First variables with the same domain have the same color. Similar constraints (i.e. constraints defined by the same relation) have the same color. For each constraint, the binding-nodes corresponding to locally symmetric variables have the same color. The same coloring schema of binding-nodes is used for similar constraints. In all other cases, colors must be different.

We can show that the above construction is correct: any automorphism in the lsv-graph corresponds to a variable symmetry in the CN. Indeed, any element (domains, constraints) constraining the search space of the CN is taken into account when building the structure of the graph and assigning colors.

As an illustration, let us consider a CN P involving a set of four variables $\{X_0 \dots X_3\}$ and a set of four intensional constraints $\{C_0 \dots C_3\}$. The associated predicate of both C_0 and C_1 is $|\$1 - \$0| = 56$ while it is $|\$1 - \$0| > 42$ for C_2 and C_3, where $\$i$ denotes the i^{th} formal parameter of the predicate. Figure 2 depicts the lsv-graph built from P: the four white nodes correspond to variable-nodes (as we assume here that they have the same domain) and two grey nodes and two black nodes correspond to constraint-nodes (grey ones for C_0 and C_1 and black ones for C_2 and C_3). Each constraint is linked to two variable-nodes through two binding-nodes since constraints are binary.

Running Algorithm 1 on C_0, variables X_0 and X_1 are detected as locally symmetric for C_0. As a consequence, the two binding-nodes introduced for C_0 receive the same

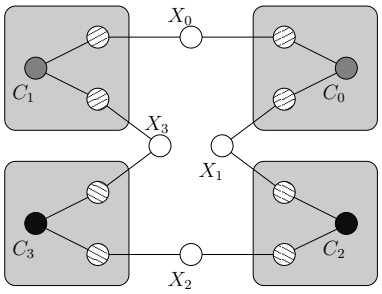


Figure 2. Illustration of lsv-graph

color: it is represented here with a right-hatched pattern. On our example, C_1 is similar to C_0 (the underlying relation is the same since both constraints are represented by the same predicate expression). This is why the same set of colors is used for C_0 and for C_1. The same principle applies to constraints C_2 and C_3: all binding-nodes for these two constraints are assigned the same color, represented here by a left-hatched pattern. Using Saucy, a generator for the symmetry group is identified: it maps X_1 into X_3, and vice-versa. This is what can be observed in Figure 2.

An advantage of lightweight detection of variable symmetries is the controlled size of lsv-graphs.

Theorem 3 *Let P be a CN. The number of nodes and edges of the lsv-graph built from P is $O(er)$ where r denotes the greatest constraint arity.*

Although some variable symmetries correspond to interchangeable variables, locally symmetric variables are not necessarily interchangeable. Moreover, variable symmetries identified via lsv-graphs cannot always be expressed in terms of interchangeable variables. For the $4\text{-}queens$ instance, there is only one variable symmetry (other than identity). This is $f^h = \{(X_a, X_d), (X_b, X_c)\}$ which is obtained automatically from the lsv-graph generated for $4\text{-}queens$. The reader can check that there is no pair of interchangeable variables for this instance.

It is important to relate this approach to that described in [14] when restricted to the identification of variable symmetries. The size of generated lsv-graphs is $O(er)$ whereas the size of Puget's graphs (and of full assignments graphs in [11]) grows exponentially with the arity of the constraints when constraints are defined in extension and when the size of the tables is not bounded, i.e. is $O(ed^r)$ where d denotes the greatest domain size. For example, let us consider the non-binary instance $steiner\text{-}7$ [11]. On the one hand, the generated Puget's graph contains $347,760$ nodes and $1,032,689$ edges and the full assignments graph contains $154,294$ nodes and $459,557$ edges. On the other hand, the lsv-graph only contains 231 nodes and 336 edges while allowing us to detect the full set of interchangeable variables. Recall here that we only focus on the detection of variable symmetries. However, the size of generated lsv-graphs renders our approach practical on very large instances.

When constraints are binary, these different approaches detect the same groups of variable symmetries, but this is not always true for non-binary constraints. For example, the constraint C such that $scp(C) = \{X_1, X_2, X_3, X_4\}$ and $rel(C) = \{(4,3,2,1),(1,2,3,4)\}$ admits a variable symmetry σ such that $\sigma(X_1) = X_4$, $\sigma(X_2) = X_3$, $\sigma(X_3) = X_2$ and $\sigma(X_4) = X_1$, but no locally symmetrical variables. This is a symmetry composed of two cycles, similar to that found for $4\text{-}queens$. At the level of a single constraint, an lsv-graph cannot handle this. It would be worthwhile to extend lightweight detection of symmetries to deal with locally symmetrical groups of variables (that is, a generalization of locally symmetrical variables), while controlling the time complexity of local symmetry detection and the space complexity of generated lsv-graphs.

4. Conclusion

In this paper, we have introduced a new graph representation of CNs based on the identification of locally symmetric variables. This representation is made homogeneous by disregarding the representation of constraints, and the size of the obtained lsv-graphs is linear w.r.t. the number of constraints (and their arity). Inspired from both transformation rules [9] and graph automorphism identifications [4, 15, 14], our approach automatically detects variable symmetries.

References

[1] F. Aloul, K. Sakallah, and I. Markov. Efficient symmetry breaking for Boolean satisfiability. *IEEE Transactions on Computers*, 55(5):549–558, 2006.

[2] R. Backofen and S. Will. Excluding symmetries in constraint based search. *Constraints*, 7(3-4):333–349, 2002.

[3] D. Cohen, P. Jeavons, C. Jefferson, K. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. *Constraints*, 11(2-3):115–137, 2006.

[4] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proceedings of KR'96*, pages 148–159, 1996.

[5] P. Darga, M. Liffiton, K. Sakallah, and I. Markov. Exploiting structure in symmetry generation for CNF. In *Proceedings of DAC'04*, pages 530–534, 2004.

[6] T. Fahle, S. Schamberger, and M. Sellman. Symmetry breaking. In *Proceedings of CP'01*, pages 93–107, 2001.

[7] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Proc. of CP'01*, pages 77–92, 2001.

[8] E. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of AAAI'91*, pages 227–233, 1991.

[9] A. Frisch, I. Miguel, and T. Walsh. CGRASS: A system for transforming constraint satisfaction problems. In *Proceedings of CSCLP'02*, pages 23–36, 2002.

[10] I. Gent and B. Smith. Symmetry breaking during search. In *Proceedings of ECAI'00*, pages 599–603, 2000.

[11] Y. Law, J. Lee, T. Walsh, and J. Yip. Breaking symmetry of interchangeable variables and values. In *Proceedings of CP'07*, pages 423–437, 2007.

[12] C. Lecoutre. *Constraint networks: techniques and algorithms*. ISTE/Wiley, 2009.

[13] B. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30:45–87, 1981.

[14] J. Puget. Automatic detection of variable and value symmetries. In *Proceedings of CP'05*, pages 475–489, 2005.

[15] A. Ramani and I. Markov. Automatically exploiting symmetries in constraint programming. In *Proceedings of CSCLP'04*, pages 98–112, 2004.