Alastair F. Donaldson Peter Gregory
Karen E. Petrie (Eds.)

Symmetry and Constraint Satisfaction Problems

Sixth International Workshop
Cité des Congrès, Nantes, France, 25th September 2006
Proceedings

Held in conjunction with the
Twelfth International Conference on
Principles and Practice of
Constraint Programming (CP 2006)

Preface

This volume contains a selection of papers on the topic of symmetry breaking for constraint programming, marking the sixth in a successful series of workshops on the topic. SymCon'06 follows earlier workshops at CP '01 in Paphos, Cyprus, at CP '02 in Ithaca, NY, USA, at CP '03 in Cork, Ireland, at CP '04 in Toronto, Ontario, Canda, and at CP '05 in Barcelona, Spain.

Of the eight papers which make up these proceedings, the first six were invited for presentation at the workshop. The contributions investigate a range of problems relating to symmetry breaking. Mears et al. present an approach to implementing symmetry detection techniques proposed by Puget in a paper at last year's CP conference. Benhamou and Saïdi extend techniques from a paper by Benhamou in the proceedings of SymCon '04 for exploiting symmetry in *not-equals binary constraint networks*. The use of comutational group theoretic algorithms to generalise static symmetry breaking is the topic of a contribution by Jefferson et al. There are two papers on breaking *almost-symmetries* in CSPs, by Verroust and Prcovic, and Martin respectively. Martin's work builds on his notion of *weak symmetry* which appeared in the proceedings of SymCon '04. A symmetry breaking technique for subgraph pattern matching is presented by Zampelli et al. Heller and Sellmann investigate the relationship between symmetry breaking and random restarts. The final paper in the proceedings compares the *symmetry breaking during search* technique with the *dynamic lex constraints* technique (to be presented at CP '06).

We would like to thank all the authors who submitted papers, the members of the Programme Committee, and the CP '06 Workshop and Tutorial chair Barry O'Sullivan. We also thank Chris Jefferson for help in reviewing papers.

These proceedings can be found online at `http://www.cis.strath.ac.uk/~pg/symcon06/`. After the conference, we hope to publish a selection of these papers in a volume dedicated to the CP '06 workshops.

August 2004

Alastair F. Donaldson
Peter Gregory
Karen E. Petrie
(Programme Chairs)

Programme Committee

Workshop Schedule

08:55 - 09:00 — Welcome

09:00 - 09:30 Reasoning by Dominance in Not-Equals Binary Constraint Networks
Belaïd Benhamou and Mohamed Réda Saïdi 9

09:30 - 10:00 On Implementing Symmetry Detection
Christopher Mears, Maria García de la Banda and Mark Wallace 1

10:00 - 10:30 Solving Partially Symmetrical CSPs
Florent Verroust and Nicolas Prcovic 24

10:30 - 11:00 — Coffee break

11:00 - 11:30 GAPLex: Generalised Static Symmetry Breaking
Chris Jefferson, Tom Kelsey, Steve Linton and Karen Petrie 17

11:30 - 12:00 Symmetry Breaking in Subgraph Pattern Matching
Stéphane Zampelli, Yves Deville and Pierre Dupont 31

12:00 - 12:30 Dynamic Symmetry Breaking Restarted
Daniel S. Heller and Meinolf Sellmann 39

— Close and lunch

Contents

On implementing symmetry detection

C. Mears and **M. Garcia de la Banda** and **M. Wallace**
Clayton School of IT, Monash University, 6800, Australia
{cmears,mbanda,wallace}@mail.csse.monash.edu.au

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 [9]. In particular, it proposes a modification of the approach to deal with general expressions, discusses the insights gained, and gives the results of a preliminary experimental evaluation of the accuracy and efficiency of the approach.

1 Introduction

Automatic symmetry detection in constraint satisfaction problems (CSP) is a significant issue. This is mainly due to the fact that symmetries can be used to speed up the search for solutions. Once a solution is found, symmetries can be applied to quickly derive its symmetric solutions. Also, areas of the search space can be ignored if they are symmetric to dead ends already explored.

A common classification of symmetries distinguishes among *variable* symmetries, *value* symmetries, and *variable-value* symmetries, which refer to permutations among the variables in the CSP, among the values of each variable, and among variable-value pairs, which preserve the set of solutions [8].

The importance of automatically detecting symmetries has generated a considerable amount of interest and has resulted in many different methods that detect one or more kinds of symmetries (e.g., [5, 6, 10, 9]). One of the most promising techniques is to construct a graph and use graph automorphism to detect symmetries. One such method is presented in [10] to detect variable symmetries by specifying constraints in a high-level format based on unsigned integer variables and mathematical operators. Symmetries are then detected using their parse graph as input to the automorphism finder program Saucy [3] which produces the generators for the symmetry group.

Our paper presents work in progress performed while studying and implementing the method presented in [9], which can be seen as an extension of [10]. The method was selected because it can deal not only with variable symmetries, but also with value symmetries and variable-value symmetries other than those obtained by the simple combination of the two.

During the implementation of the method, several issues, such as how to deal with general expressions, had to be resolved. This led not only to modifications to the original method, but also to interesting insights into it. In this paper we present the modifications performed, discuss the insights gained, and give the results of a (very) preliminary experimental evaluation of the accuracy and efficiency of the approach.

2 Background

This section introduces the terminology to be used in the paper and provides a brief summary of the method presented in [9] to automatically detect symmetries.

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, each variable $x_i \in X$ is associated with a finite domain $D_i \in D$ of potential values, and each constraint $c \in C$ over distinct variables $x_i, \ldots, x_j$ specifies the allowed subset of the Cartesian product $D_i \times \ldots \times D_j$ (i.e., the values consistent with the constraint).

A *literal* has the form $x_i = d$ where $d \in D_i$. An *assignment* for constraint c defined over variables $x_i, \ldots, x_j$, is a tuple $\langle d_i, \ldots, d_j \rangle$ where $d_i \in D_i, \ldots, d_j \in D_j$.

A *solution symmetry* for a CSP is a bijection from literals to literals that preserves the set of solutions of the problem. A *constraint symmetry* is an operation that preserves its constraints. While any constraint symmetry is also a solution symmetry, a CSP may have many solution symmetries that are not constraint symmetries [1]. A *variable symmetry* is a permutation of the set X (i.e., a bijection from variables to variables) that preserves the set of solutions. 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 set of solutions [8].

2.1 Puget's approach

The method presented by Puget in [9], is based on two steps. The first uses the CSP to construct a coloured graph (V, E, c) where V is a set of nodes, E a set of unweighted and undirected edges, and c a map from V to colours. The second step finds the automorphisms

of the graph, i.e., a one to one mapping $f : V \to V$ such that $\forall (n_i, n_j) \in E : (f(n_i), f(n_j)) \in E$ and $\forall n \in V, c(f(n)) = c(n)$. Colours are thus used to restrict the automorphisms of the graph, since only nodes with the same colour can be interchanged.

The graph associated with the CSP (X, D, C) is constructed as follows. For every variable $x_i \in X$ and every value $d \in D_i$, a *literal* node is created representing the literal $x_i = d$.

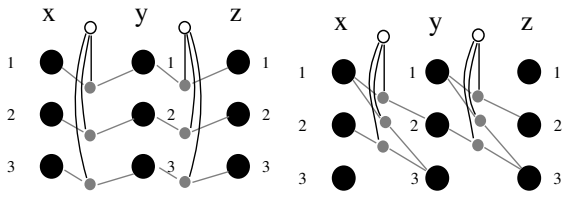
Example 2.1 Consider a variable x_i with domain $D_i = \{1, 2, \ldots, 10\}$. It is represented by ten literal nodes corresponding to $x_i = 1$, $x_i = 2$, ..., $x_i = 10$. □

For every constraint $c \in C$ defined over distinct variables $x_i, \ldots, x_j$, a *constraint* node c is created. Also, for every assignment $\langle d_i, \ldots, d_j \rangle \in c$, an *assignment* node is created and connected to each literal node $x_i = d_i, \ldots, x_j = d_j$ and to the constraint node c. If the subset of the Cartesian product specified by c (i.e., the set of consistent assignments) is much bigger than $(D_i \times \ldots \times D_j) \setminus c$ (i.e., the set of inconsistent assignments), then the latter can be used to construct an equivalent but smaller graph.

Once the nodes and edges are determined, the last step in the construction of the graph is the mapping of nodes to colours. All literal nodes in the graph are mapped to the same colour, which is different from any non-literal node in the graph, i.e., if $V_L \subset V$ denotes the set of literal nodes, $\forall n_i, n_j \in V_L : c(n_i) = c(n_j)$ and $\forall n \in (V \setminus V_L) : c(n) \neq c(n_j)$. This allows the detection of variable symmetries, value symmetries and value-variable symmetries. Note that if the colour given to the literal nodes of each variable was unique (i.e., if $\forall n_i, n_j \in V$ representing literals $x_i = d_i$ and $x_j = d_j$ we have that $c(n_i) = c(n_j) \iff x_i \equiv x_j$), then only value symmetries would be detected. Similarly, if the colour given to the literal nodes of each value was unique (i.e., if $\forall n_i, n_j \in V$ representing literals $x_i = d_i$ and $x_j = d_j$ we have that $c(n_i) = c(n_j) \iff d_i \equiv d_j$), then only variable symmetries would be detected.

Regarding constraint and assignment nodes, all assignment colours for a given constraint are given the same colour which is different from those used for literal and constraint nodes. Also, the constraint node of each constraint is mapped to yet another colour. Puget also indicates that in order to enable constraint symmetries, "all constraints of the same kind have the same colour." We interpret this from an intensional point of view, in which constraints are of the form $f(x_i, \ldots, x_j)$, where f is the constraint name and $x_i, \ldots, x_j$ are distinct variables. Then, constraints with the same f and number of arguments are considered of the "same kind". In order to detect constraint symmetries, we must not only merge the colour of their constraint nodes, but also that of their assignment nodes.

Example 2.2 Consider a CSP with set of variables $\{x, y, z\}$, where each variable has domain $\{1, 2, 3\}$, and the set of constraints is (the extensional representation of) $\{x = y, y = z\}$. This CSP results in the graph shown in Figure 1(a). The graph has three literal nodes per variable (one for each value in their domains), all mapped to the same colour (black). It also has three assignment nodes corresponding to each allowed assignment $\langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 3 \rangle$ of $x = y$, and another three for those of $y = z$. Since both constraints have the same functor/arity (=/2), all their assignment nodes are mapped to the same colour (grey). Finally, the graph has two constraint nodes, each connected to the associated assignment nodes and both mapped to the same colour (white). The graph associated with a similar CSP with constraints $\{x < y, y < z\}$ is shown in Figure 1(b). □



(a) $x = y, y = z$ (b) $x < y, y < z$

Figure 1: Graphs before applying transitive closure.

While the method described above is correct (any symmetry of the graph is a symmetry of the CSP), it is not complete (some CSP symmetries might not appear in the graph). This was demonstrated by Puget in [9] using the graph of Figure 1(a) which represents constraints $\{x = y, y = z\}$. The graph does not contain any variable symmetries involving y, even though they exist. Puget thus suggested to take the transitive closure of equality constraints, that of $\leq$ constraints, and also to replace constraints such as $x \leq y$ and $y \leq x$ by $x = y$.

Example 2.3 Applying this to the graph of Figure 1(a) leads to the addition of the constraint $x = z$, and results in the graph shown in Figure 2. This graph does contain the variable symmetries relating x and y, and z and y. □

Note that the above discussion is based on constraints whose arguments are distinct variables. Puget indicates that any expression of the form $x_i \ op \ x_j$, where $x_i, x_j \in X$ are distinct variables, can be treated as the extensional constraint associated to $op(x_i, x_j, w)$, where w is a new variable. However, he indicates this will lead to very large graphs and thus proposes to handle only expressions of the form $f(x)$, where the variable x is only allowed to occur once in the expression. No extra nodes are used to represent these expressions; rather, the literal

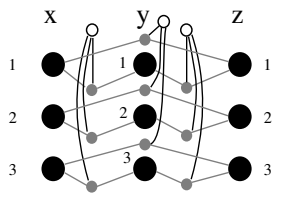


Figure 2: Transitive closure on equality.

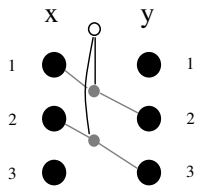


Figure 3: $x + 1 = y$

node $x = d$ representing the value d is used to compute the value for the expression. Thus, the constraint is represented in extension.

Example 2.4 Figure 3 shows the graph associated to a CSP with the single constraint $x + 1 = y$. The literal node $x = 1$ plays the role of $x + 1 = 2$, and $x = 2$ plays the role of $x + 2 = 3$.

3 Implementing Puget's approach

We have implemented Puget's method for ECL^{i}PS^e programs which use the `ic` and `ic_sets` constraint libraries. In order to do so, we wrote a new ECL^{i}PS^e library that mimics (part of) the interfaces of the `ic` and `ic_sets` libraries. When the program is executed in ECL^{i}PS^e, the constraint predicates in the library are called (rather than those defined by the associated solver).

These predicates output the intensional constraints generated during the execution of the CSP into a file in simple text format. This file is in turn processed by a graph generator that converts the intensional constraints in the file into their graph representation. The resulting graph is finally input to a graph automorphism package which returns the generating automorphisms.

While the graph generation could be performed directly by the new library interface, as we will see later, there are advantages in producing the intermediate text format, especially for a prototype which aims to explore different alternatives for constructing the graph.

This section describes the problems encountered while writing the new library and the resulting additions and/or modifications devised to solve them. In doing so it focuses on representing expressions other than $f(x)$.

Note that the graphs built by our system do not contain constraint nodes and, thus, none of the example graphs in the following sections contain these nodes either. The significance of this will become clear later on.

3.1 First attempt at expressions

Our first attempt at handling general expressions followed the previously introduced idea of representing any arbitrary operation x_i *op* x_j by creating a new variable w and representing instead the new ternary constraint $op(x_i, x_j, w)$. Expressions with more than one operator are broken into sub-expressions, each of which is represented by a new (temporary) variable t_i. For example, $x + y - z$ is parsed as $(x + y) - z$ and is represented by two new variables: $t_1 = x + y$ and $t_2 = t_1 - z$. Constraints involving expressions as arguments are simply treated by replacing each such argument by a new variable representing the expression. Constant arguments are represented by a new variable with a single literal node whose value is that of the constant.

In this approach care must be taken with associative operators [9, 10].

Example 3.1 The expression $x + y + z$, which is parsed as $(x + y) + z$, would result in two new variables: $t_1 = x + y$ and $t_2 = t_1 + z$. But if so, the only variable symmetry in the associated graph is $x \leftrightarrow y$, whereas in reality all three variables are interchangeable. □

This problem is ameliorated by pre-processing the text format output by our library to group associative operators as follows: any "dummy" variable t_i that is used in an expression (e.g. $t_1 + z$) is replaced by its definition (e.g. $x + y$) if the operators in the expression and definition are the same (both addition in this example) and associative. Thus, the expression becomes $t_2 = x+y+z$, preserving the symmetries.

As recommended in [9], non-symmetric binary arithmetic operands, 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.

Example 3.2 Consider the equation $(a + b) * (c + d) * (e - f)$, where all variable domains are 1..3. The expression $e - f$ is represented as $e + (-f)$ which ensures that e and f are not symmetric. It also prevents $(e - f)$ from being interchangeable with $(c + d)$. The following symmetries remain: $a \leftrightarrow b$, $c \leftrightarrow d$, $(a+b) \leftrightarrow (c+d)$, and the variable-value symmetry $(e = 1 \leftrightarrow f = 3,\ e = 2 \leftrightarrow f = 2,\ e = 3 \leftrightarrow f = 1)$. □

Intermediate variables can, however, prevent some symmetries from being captured by the graph.

Example 3.3 Consider a CSP with variables $\{x, y, z\}$ where $D_x = D_y = \{1, 2\}$, $D_z = \{1\}$ and the constraints are $x + z \neq y \wedge y + z \neq x$. This CSP has a variable symmetry ($x \leftrightarrow y$) and a value symmetry (values 1 and 2 are interchangeable). The expressions in the two constraints are represented by $t_1 = x + z$ and $t_2 = y + z$. The associated graph is shown in Figure 4, with grey and white nodes representing assignment nodes for equality and disequality constraints, respectively. It can be seen that the graph captures the variable symmetry, but not the value symmetry. □

Example 3.4 The constraints in the above example appear in some of the models for the n-queens problem (with variable z replaced by constant 1). Figure 5 shows

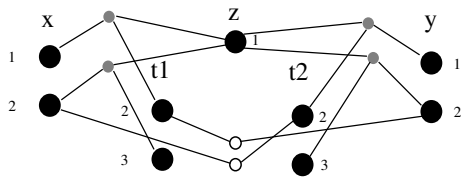


Figure 4: $x + z \neq y$ and $y + z \neq x$, $D_x = D_y = \{1, 2\}$, $D_z = \{1\}$

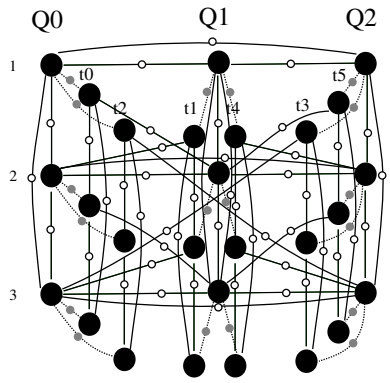


Figure 5: 3-queens using the first method

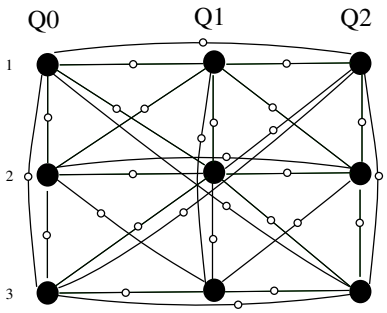


Figure 6: 3-queens using the second method

the graph for 3-queens where the parsed constraints are $Q0 \neq Q1, Q0 \neq Q2, Q1 \neq Q2, t_0 \neq Q1, t_1 \neq Q0, t_2 \neq Q2, t_3 \neq Q0, t_4 \neq Q2, t_5 \neq Q1$ where $t_0 = Q0 + 1$, $t_1 = Q1 + 1$, $t_2 = Q0 + 2$, $t_3 = Q2 + 2$, $t_4 = Q1 + 1$, $t_5 = Q2 + 1$. Grey and white nodes represent equality and disequality assignments, respectively. For visual clarity, the graph omits the constant summands and indicates edges of equality constraints as solid, and of disequality constraints as dashed. Again, only the variable symmetry ($Q0 \leftrightarrow Q2$) is captured. □

Obviously, the $f(x)$ representation can easily be used to avoid the problem in n-queens, since this representation would avoid the creation of intermediate variables, by absorbing the constant 1 into the value of x. However, the example is useful since it allows us to show the problem in a very well known setting.

The fact that the constraint syntax bears so much influence on the resulting graph, highlights the importance of normalisation. 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 at this point that we decided to abandon the above method, which not only generated too many intermediate variables (as already indicated by Puget), but also could easily result in symmetries being missed due to the use of a particular syntax.

3.2 Second attempt at expressions

Our solution was to go back to basics, i.e., to rethink the issue in terms of the extensional representation of the constraint, rather than their intensional representation (which relates too closely to the syntax). 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. This approach also leads to huge graphs.

Example 3.5 The constraint $u = 2 * x + (3 * y * 7 * z)$ does not need to be split up into different components and can simply be represented by the original method by considering its extensional meaning, i.e., the consistent subset of $D_x \times D_y \times D_z \times D_u$. □

This leads to a much simpler graph in which each constraint directly appearing in the model is associated with a number of assignment nodes (one per assignment allowed by the constraint). Each assignment node is linked to the set of variable nodes representing the value associated by the assignment to each variable.

Example 3.6 Figure 6 shows the 3-queens obtained using this method. Note that assignments for disequality constraints are shown as white nodes. All the symmetries of the chessboard are now captured. □

The correctness of this method follows directly from the correctness of Puget's method, since it simply applies the extensional meaning of constraints.

We find the simplicity of this method very pleasing since it eliminates problems such as the representation of constants (which is now avoided for any constant regardless of whether 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), the problem of associative and non-symmetric operands appearing in the same constraint, and in general, any such normalisation issue.

The method can easily be applied to any finite domain. Let us consider, for example, how it would be applied to CSPs with set variables. If a set variable $x_i \in X$ is a subset of the set S, its domain D_i is equal to the power-set of S. As a consequence, a variable that is the subset of S is represented by $2^{|S|}$ literal nodes.

Example 3.7 Consider a CSP where x_i is defined as a subset of $\{1, 2, 3, 4\}$. The graph will thus contain 16 variable nodes representing the literals $x_i = \{\}$, $x_i = \{1\}$, $x_i = \{2\}$, ..., $x_i = \{2, 3, 4\}$, $x_i = \{1, 2, 3, 4\}$.

Implemented set constraints include equality, disequality, cardinality, and all_disjoint. Implemented expressions include union and intersection.

Example 3.8 Consider a cardinality constraint of the form $|x_i| = x_j$, where x_i is a set variable and x_j is an integer variable. Then, for every node $n_i \in V$ corresponding to value $d_i \in D_i$ for which there exists a node n_j corresponding to value $d_j \in D_j$ such that $|d_i| = d_j$, an assignment node is created and linked to both n_i and n_j. If the cardinality constraint is of the form $|x_i| = I$,where I is an integer constant, assignment nodes will only be created for values $d_i \in D_i$ for which $|d_i| = I$. Figure 7 shows an example of a cardinality constraint on expression $(s_1 \cap s_2) \cup s_3$. □

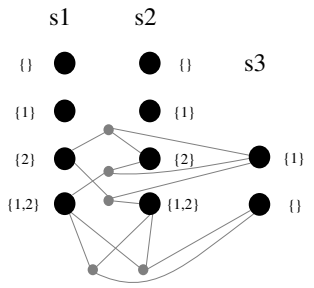


Figure 7: Graph for constraint $|(s_1 \cap s_2) \cup s_3| = 2$

A different representation can be obtained by representing each set variable x_i defined as a subset of set S, with $2 * |S|$ literal nodes, half of them representing each element in S as appearing in x_i, and the other half as not appearing in x_i. Such representation would reduce the number of nodes and increase the number of edges. The decision to use one or other approach could then depend on the particular constraints of the problem and on the complexity behaviour of the automorphism algorithm.

4 Discussion

Once we decided the particular method to be used, we focused on several issues including how to make it faster and more complete. This section discusses some of them.

4.1 Relationship with CSP microstructure

During the initial evaluations of N-Queens we discovered our program was not finding certain value-variable symmetries (corresponding to diagonal symmetries). The reason was the lack of the disequality constraints implicit among the values of every variable *in every CSP*. This means that upon creation of the literal nodes associated to a variable, assignment nodes and associated edges representing such constraints must be created. This is, of course, the case regardless of the representation method chosen for expressions.

At this point the similarities between the resulting graph and the microstructure and microstructure complement described in [1] became apparent. A CSP's microstructure is a hypergraph with a node for each value in the domain of each variable, and a hyperedge corresponds to 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. The automorphisms of this graph contain all constraint symmetries in the associated CSP [1].

Our graph has also a node for every possible value of each variable, and contains information about consistent and inconsistent assignments. However, hyperedge $\langle v_1, \ldots, v_n \rangle$ would be represented in our graph by assignment node a and the edges $\langle a, v_1 \rangle, \ldots, \langle a, v_n \rangle$.

We are currently studying how to use this relationship to improve our system. For example, note that the microstructure has no notion of coloured nodes. This would seem to indicate we do not need them either. This is actually the case if we used only consistent (or only inconsistent) assignments. Otherwise, colours (but only two) are needed.

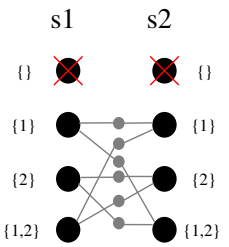


Figure 8: Pruning unnecessary values.

4.2 Pruning inconsistent literal nodes

Even without temporary variables and constants, representing constraints extensionally leads to huge graphs. Generally, the size of the graph increases exponentially in the arity of the CSP's constraints. Therefore, there is a practical limit on the size of problem instance that can be handled by the method. We propose to increase this limit by "pruning" unnecessary components from the graph, in particular, any literal node (and associated edges) corresponding to a literal $x_i = d$, if no allowed tuple for c assigns value d to variable x_i. Since such a node cannot be part of a solution, it can simply be removed from the graph.

Example 4.1 Consider a CSP with two set variables, s_1, s_2 where $s_1, s_2 \subseteq \{1, 2\}$ (i.e. $D_{s_1} = D_{s_2} = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$), and the single constraint $|s_1 \cap s_2| = 1$. The graph of this CSP is shown in Figure 8. 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 to these literals can thus be removed from the graph. □

A literal node that is not used in any allowed tuple of a constraint (as in the empty set values in Figure 8) may yet be part of some allowed tuple in another constraint. Pruning of such unviable nodes can then be propagated as follows: when an unviable literal node n is removed, any edges and assignment nodes from other constraints that involve n can also be removed. This in turn may cause other literal nodes to become unviable. This process can continue iteratively until no more edges or nodes can be removed.

Example 4.2 Consider a CSP similar to that of Example 4.1 but with three set variables, s_1, s_2, s_3 where $s_1, s_2, s_3 \subseteq \{1, 2\}$ (i.e. $D_{s_1} = D_{s_2} = D_{s_3} = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$), and two constraints, the previous one $|s_1 \cap s_2| = 1$ and the constraint $|s_2 \cap s_3| = 0$. The graph of this CSP is shown in Figure 9. When the literal node associated to literal $s_2 = \{\}$ is removed, the four assignment nodes marked with crosses are also removed. With those assignments gone, the literal nodes $s_3 = \{1\}$, $s_3 = \{2\}$ and $s_3 = \{1, 2\}$ are no longer viable an can be removed. □

	Edges	Nodes	Assign
golf222	464/2810	206/1020	150/946
golf322	1290/7629	548/2681	423/2559
steiner5	333/5898	157/2080	102/1977
steiner6	6215/65830	2226/22321	2035/21980

Table 1: Size of the graphs obtained for social golfers and Steiner triplets with/without pruning.

Of course, care needs to be taken when inconsistent assignments have been used to build the graph. Inconsistent values should only be considered for removal for constraints which use consistent assignments, and vice versa. Furthermore, while pruning might prevent some symmetries from being detected, it might also lead to the detection of symmetries that are in the CSP but not captured in the initial graph. We are currently studying the effects of different pruning strategies to this.

Preliminary results on the effect of pruning are promising. We have implemented pruning for the first method on cardinality constraints. These constraints are present in the social golfers and Steiner triplet models, and pruning reduces their graphs dramatically (see Table 1).

While pruning can help reduce the graph, it cannot overcome the size problem. This is however sufficient for our long-term project, which is to be able to infer symmetries for a CSP problem (rather than for a particular instance of the problem) from examining relatively small instances of that problem.

4.3 Implied constraints

While the correctness of the method is clear, its completeness is a different matter. It is easy to prove that given a CSP (X, D, C), all symmetries contained in a given constraint $c \in C$ are present in the graph built from an equivalent CSP containing only c. While this alone is quite pleasing, it does not address the issue of the conjunction of the constraints.

This is related to the problems addressed in, for example, [9] by means of a transitive closure applied to equality and inequality constraints in the CSP. This method can be generalised to the detection and addition of implied (also referred to as entailed) constraints. Since, in general, such detection can be complex, the question is which implied constraints would help detect symmetries that otherwise would have gone undetected.

We are currently considering the use of abstract interpretation techniques [2] to infer such information from the CSP model. Note, however, that any automatic analysis of the constraints aimed at inferring implied constraints can be based on the original set of (intensional) constraints represented in the program, or on the associated graph. Indeed, some properties (such as variable aliasing) might be easier to infer from one than from the other. Since, however, the graph represents a particular instance of the problem (i.e., the original model instantiated for a given set of values), the properties inferred from it might only be applicable to that instance, rather than to the original model.

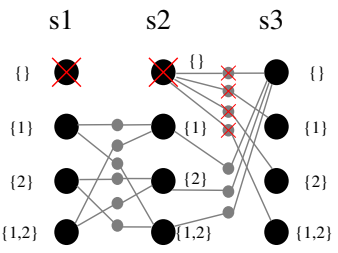


Figure 9: Pruning unnecessary values.

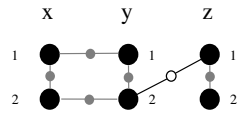


Figure 10: Projection of a variable.

Finally, note that the issue of implied constraints is also closely related to the issue of k-ary nogoods in [1], where it is proved that if all are present in the graph of a k-ary CSP, then the graph is known to capture all solution symmetries in the CSP (rather than only the constraint symmetries).

This relationship makes the following property clear: if a constraint c among variables $x_i, \ldots, x_j$ can be equivalently expressed as the conjunction of a set of constraints, each of them defined over a strict subset of $x_i, \ldots, x_j$, then it is advantageous to use the set of constraints rather than c.

Example 4.3 Global constraint `alldifferent`$(x_1, \ldots, x_k)$ should be expressed as the set of constraints $\{x_i \neq x_j | 1 \leq i < j \leq k\}$.

4.4 Projecting out variables

We would like to be able to easily eliminate variables from the graph of a CSP, representing the operation of projecting out such variables from the CSP. Such projection has many uses, including the partition of the problem into smaller components whenever variables have different types, and the possibility of focusing on a much smaller set of variables of interest.

One might think it is possible to project out variables by removing all literal nodes associated to that variable and their connecting edges, as long as all implied constraints are represented in the graph. However, this is incorrect as it may introduce spurious symmetries in the remaining subgraph.

Example 4.4 Consider a CSP with three variables, x, y, z all with domain 1..2 and constraints $x \neq y, z < y$. The graph is shown in Figure 10. While, the CSP has no symmetries, if variable z is projected out the graph does become symmetric.

We believe this is related to the concept of projection independence defined in [4], which was developed to automatically parallelize CLP programs. We plan to study its relationship in the future.

Instance	Graph Details			Order	Running Time in Seconds					
	Edges	Nodes	Assign		Total	Text	GGen	GPrt	SRun	HR
bibd33110	639	335	234	72	0.4	0.37	0.01	0.01	0.01	0.00
bibd610532	247505	26274	25100	15676416000	71.69	0.33	16.43	36.83	0.94	1.12
bibd77331	41461	8129	7014	50803200	9.4	0.31	2.34	4.05	0.39	0.55
golf222	11702	2532	2482	192	0.61	0.28	0.14	0.10	0.06	0.02
golf232	598869	85742	85596	51840	43.25	0.28	8.85	9.17	22.09	1.41
golf322	30267	6509	6399	1152	1.47	0.28	0.48	0.35	0.25	0.05
golf332	1486917	213554	213228	933120	205.12	0.28	33.26	42.54	110.63	5.98
golomb4	3614	1827	1615	0	0.39	0.28	0.04	0.05	0.00	0.01
golomb5	24650	11745	11045	0	1.25	0.28	0.39	0.42	0.05	0.01
golomb6	147852	68328	66246	0	8.68	0.28	2.84	4.21	0.28	0.09
latin10	27000	14500	13500	286708355039232000000	4.55	0.31	1.86	0.98	0.31	0.81
latin11	39930	21296	19965	3816088205572177920000000	7.68	0.36	3.03	1.71	0.64	1.30
latin12	57024	30240	28512	659420041922872344576000000	13.58	0.43	5.06	3.47	0.99	2.35
latin13	79092	41743	39546	144874583210455054103347200000	22.09	1.23	8.48	5.07	1.64	3.73
latin14	107016	56252	53508	39753585632948866845958471680000000	60.21	25.8	13.03	9.41	2.46	5.93
queens8	5488	3256	2744	2	0.58	0.28	0.11	0.13	0.02	0.02
queens10	12840	7420	6420	2	1.24	0.31	0.28	0.43	0.03	0.05
queens20	184680	100340	92340	2	51.49	0.31	5.62	29.05	0.59	3.57
steiner4	2151	469	454	48	0.33	0.28	0.03	0.01	0.01	0.00
steiner5	29370	5086	5049	720	1.03	0.29	0.30	0.24	0.16	0.03
steiner6	436410	63051	62940	86400	23.29	1.42	5.81	6.13	8.18	1.08

Table 2: Results for the first method.

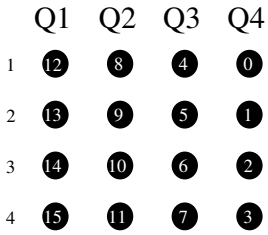


Figure 11: Literal nodes of 4-queens.

5 Experimental Evaluation

As mentioned before, we have implemented the previous methods in ECL^{i}PS^e for programs which use the `ic` and `ic_sets` solvers by building a library that mimics their interface and outputs the associated intensional constraints into a text file. This file is processed by a graph generator which builds their graph representation.

The resulting graph is input to the graph automorphism package Saucy [3] which returns the generating automorphisms.[1] Saucy describes the automorphism group generators in terms of the non-negative integers that label each node. Our system prints the generating symmetries in more intuitive form by inversing the map that associates integer labels with variable-value labels.

Example 5.1 Consider the literal nodes of 4-queens, shown in Figure 11. The mapping of the literal nodes is shown by the white numbers. 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)
```

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

Currently, these symmetries are not used to aid a search, but we intend to implement this in the near future using GAP-SBDS [7].

Tables 2 and 3 show our results. Each row in the tables corresponds to a different instance of a well known benchmark problem for symmetries. In particular, row bibdVBRKL shows the data for an instance of the Balanced Incomplete Block Design problem called with V varieties, B blocks, R size of each variety, K size of each block, and L blocks in which each two varieties appear together; row golfWGP shows the data for instances of the Social Golfers called with W weeks, G groups, and P players per group; row golombN shows the data for instances of the Golomb Ruler problem called with N integers; row latinN shows the data for instances of the Latin Square problem called with a square of size N; row mostperfectN shows the data for instances of the Most Perfect Magic Square problem called with N columns and rows; row queensN shows the data for instances of the N-Queens problem; and row steinerN shows the data for instances of the Steiner triplets problem called with the elements of the triples taking any value from 1 to N.

The columns in both tables are as follows: Edges, number of edges in the graph; Nodes, total number of nodes; Assign, number of those which are assignment nodes; Order, size of the discovered automorphism group; Total, total running time in seconds (numbers in parenthesis for Table 3 indicate the time taken by the first method, divided by that taken by the second); Text, time spent producing the text file; GGen, graph generation time including the computation of the extensional constraints; GPrt, time spent printing the graph to be input to Saucy; SRun, Saucy running time; and HR, time taken by reading Saucy's output information and printing our human readable form. Running times were

[1] We would have liked to have used AUTOM [9] for finding automorphisms but it is not publicly available.

Instance	Graph Details			Order	Running Time in Seconds					
	Edges	Nodes	Assign		Total	Text	GGen	GPrt	SRun	HR
bibd33110	207	113	75	72	0.3(0.75)	0.28	0.01	0.00	0.01	0.00
bibd610532	25290	3620	3197	15676416000	3.4(0.05)	0.33	1.14	1.26	0.08	0.13
bibd714632	328790	25777	24990	2636271525888000	81.95	0.48	25.02	37.38	1.23	0.92
bibd77331	6615	1815	1421	50803200	1.38(0.15)	0.31	0.51	0.28	0.03	0.12
golf222	4022	972	946	192	0.38(0.62)	0.28	0.05	0.02	0.02	0.00
golf232	138069	24212	24156	51840	6.51(0.15)	0.31	1.73	1.72	2.18	0.40
golf322	11067	2609	2559	1152	0.6(0.41)	0.28	0.15	0.10	0.03	0.02
golf332	380997	65882	65772	933120	25.39(0.12)	0.28	6.19	6.38	10.22	1.31
golomb4	1190	585	511	2	0.41(1.05)	0.34	0.05	0.01	0.01	0.00
golomb5	8250	3755	3525	2	0.55(0.44)	0.28	0.12	0.10	0.02	0.01
golomb6	48732	21303	20646	2	2.35(0.27)	0.28	0.94	0.77	0.14	0.07
golomb7	263718	112462	110691	2	16.37	0.33	6.74	6.38	1.21	0.65
latin10	27000	14500	13500	286708355039232000000	4.65(1.02)	0.32	1.85	0.99	0.32	0.86
latin11	39930	21296	19965	3816088205572177920000000	7.9(1.03)	0.36	2.93	1.87	0.57	1.46
latin12	57024	30240	28512	659420041922872344576000000	12.93(0.95)	0.45	5.11	2.93	1.01	2.40
latin13	79092	41743	39546	144874583210455054103347200000	22.2(1.00)	1.27	8.57	4.97	1.66	3.82
latin14	107016	56252	53508	39753585632948866845958471680000000	59.74(0.99)	26.2	13.16	8.24	2.50	6.12
mostperfect4	215744	56146	55888	32	13.4	0.32	7.69	3.54	1.02	0.29
queens8	1456	792	728	8	0.36(0.62)	0.28	0.05	0.02	0.00	0.01
queens10	2940	1570	1470	8	0.44(0.35)	0.28	0.10	0.04	0.01	0.01
queens20	25080	12940	12540	8	2.44(0.05)	0.36	1.41	0.46	0.07	0.07
queens30	86420	44110	43210	8	10.83	0.35	7.52	1.97	0.33	0.27
queens40	206960	105080	103480	8	36.49	0.45	25.36	6.78	0.94	1.05
steiner4	871	209	198	48	0.3(0.91)	0.28	0.01	0.01	0.00	0.00
steiner5	10170	1999	1977	720	0.52(0.50)	0.29	0.10	0.08	0.02	0.02
steiner6	129210	22031	21980	86400	5.34(0.23)	1.33	1.60	1.56	0.42	0.30
steiner7	1011416	153448	153356	25401600	48.18	0.36	17.79	18.93	4.77	3.89

Table 3: Results for the second method.

measured on a desktop with an Intel Pentium 4 3GHz CPU and 2 GB RAM, running Linux kernel 2.4.22.

For each benchmark, the symmetries found were:

- BIBD: all varieties are interchangeable; all blocks are interchangeable,
- Social golfers: all weeks are interchangeable; all groups within a given week are interchangeable; all golfers are interchangeable,
- Golomb ruler: the ruler can be reversed (180° rotation),
- Latin square: all rows are interchangeable; all columns are interchangeable,
- Most perfect magic square: geometric symmetries of a square; values can be inverted (e.g. for a 4*4 square, $1 \leftrightarrow 16$, $2 \leftrightarrow 15$, $3 \leftrightarrow 14$, etc.),
- N-queens: geometric symmetries of a square,
- Steiner triplets: all triplets are interchangeable; all values are interchangeable.

The results show that the second method outperforms the first in running time and graph size (note that some models could not be run using the first method due to memory constraints). Even so, the results also show that in both cases the graph size increases rapidly with the size of the CSP, and thus both methods are impractical for large CSP instances.

References

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

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[3] Paul T. Darga, Mark H. Liffiton, Karem A. Sakallah, and Igor L. Markov. Exploiting structure in symmetry detection for CNF. In Sharad Malik, Limor Fix, and Andrew B. Kahng, editors, *DAC*, pages 530–534. ACM, 2004.

[4] Maria Garcia de la Banda, Manuel Hermenegildo, and Kim Marriott. Independence in CLP languages. *ACM Trans. Program. Lang. Syst.*, 22(2):296–339, 2000.

[5] Eugene C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proc. AAAI'91*, volume 1, pages 227–233, 1991.

[6] Alan M. Frisch, Christopher Jefferson, and Ian. Miguel. Constraints for breaking more row and column symmetries. In Francesca Rossi, editor, *LNCS*, volume 2833, pages 318–332, 2003.

[7] Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints: Symmetry breaking during search. In Pascal van Hentenryck, editor, *LNCS*, volume 2470, pages 415–430, 2002.

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

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

[10] Arathi Ramani and Igor L. Markov. Automatically exploiting symmetries in constraint programming. In Boi Faltings, Adrian Petcu, François Fages, and Francesca Rossi, editors, *CSCLP*, volume 3419, pages 98–112, 2004.

Reasoning by dominance in Not-Equals binary constraint networks

Belaïd Benhamou and **Mohamed Réda Saïdi**
Laboratoire des Sciences de l'Information et des Systmes (LSIS)
Centre de Mathmatiques et d'Informatique
39, rue Joliot Curie - 13453 Marseille cedex 13, France
email:Belaid.Benhamou@cmi.univ-mrs.fr, saidi@cmi.univ-mrs.fr

Abstract

Dynamic detection and elimination of symmetry in constraints, is in general a hard task, but in *Not-Equals binary constraint networks*, the symmetry conditions can be simplified. In this paper, we extend the principle of symmetry to dominance in *Not-Equals Constraint Networks* and show how dominated values are detected and eliminated at each node of the search tree. A Linear time complexity algorithm which detects the dominated values is proposed. Dominance is exploited in an enumerative method adapted to solve Not-Equals CSPs. This enumerative method augmented by Dominance is experimented on both randomly generated instances of graph coloring and Dimacs graph coloring benchmarks and its performance is compared to the same method augmented by symmetry and to the well known DSATUR method. The obtained results show that reasoning by dominance improves symmetry reasoning and our method outperforms both previous methods in solving graph coloring.

1 Introduction

As far as we know the principle of symmetry is first introduced by [Krishnamurty, 1985] to improve resolution in propositional logic. Symmetry for boolean constraints is studied in [Benhamou and Sais, 1992] where the authors showed that its exploitation is a real improvement for several automated deduction algorithms' efficiency. The notion of interchangeability in CSP's is introduced in [Freuder, 1991] and symmetry in CSPs is studied in [Puget, 1993; Benhamou, 1994a]. Since that, many research works on symmetry appeared. For instance, the static approach used by James Crawford et al. in [James Crawford *et al.*, 1996] for propositional logic theories consists in adding constraints expressing global symmetry of the problem. This technique has been improved in [F.A. Aloul *et al.*, 2003] and extended to 0-1 Integer logic Programming in [F.A. Aloul *et al.*, 2004].

Since a great number of constraints could be added in the static approach, some researchers proposed to add the constraints during the search. In [I.P. Gent *et al.*, 2002], authors add some conditional constraints which remove the symmetric of the partial interpretation in case of backtrack (this technique is called SBDS). In [Fahle *et al.*, 2001; F. Focacci and M. Milano, 2001; Jean F. Puget, 2002] authors proposed to use each subtree as a no-good to avoid exploration of some symmetric interpretations (this technique is called SBDD) and the GE-trees conceptual for symmetry elimination is introduced in [Colva M. Roney-Dougal *et al.*, 2004]. More recently a method which breaks symmetries between the variables of an Alldiff constraint is studied in [Puget, 2005b], a method which eliminates all value symmetries in surjection problems is given in [Puget, 2005a], and a work gathering the different symmetry definitions is done in [Cohen *et al.*, 2005].

We investigate in this article the principle of dominance in *Not-Equals binary Constraint Networks (notation NECSPs)*. Dominance is a weak symmetry principle which extend the Full substitutability notion [Freuder, 1991]. Dominance is first introduced in [Benhamou, 1994b] for general CSPs, but it is shown that its detection is harder than symmetry. Here, we show how dominance is adapted, detected, and exploited efficiently in NECSPs. Of course, the NECSPs is a limited framework, but in theory, this restriction remains NP-complete. Indeed, Graph coloring fits in the NECSPs framework and is NP-complete [M.R. Garey and D.S. Johnson, 1979], thus solving Not-Equals CSPs is in general a NP-complete problem. Besides, in practice, this framework is quite expressive, it covers a broad range of problems in artificial intelligence, such as Time-tabling and Scheduling, Register Allocation in compilation, and Cartography [A. Ramani *et al.*, 2004].

Detecting symmetrical domain values of a CSP variable during search is in general a hard task. A symmetry detection method is proposed in [Benhamou, 1994a], but its complexity is exponential in the worst case. In case of Not-Equals CSPs, some symmetrical values are detected with a linear time complexity [Benhamou, 2004].

We show in this article how symmetry is extended to dominance in NECSPs, and how the symmetry condition given in [Benhamou, 2004] is weakened in the case of failing to instantiate a variable with a value of its domain during the search. We give a weak symmetry/dominance condition which leads to a dominance detection algorithm whose efficiency is better and which detects both the dominance and

more symmetries than the algorithm defined in [Benhamou, 2004].

The rest of this article is organized as follows: Section 2 gives a brief background on CSPs. In section 3 we discuss the dominance notion and show how the symmetry condition given in [Benhamou, 2004] is weakened and extended to dominance. We give in the subsection 3.4 an efficient dominance detection algorithm in Not-Equal CSPs. We show in section 4 how Dominance is exploited in a simplified forward checking (SFC) method adapted to Not-Equals CSPs. In section 5 we evaluate and compare the effectiveness of our result by carrying experiments on both Dimacs graph coloring benchmarks and randomly graph coloring generated instances. Section 6 discusses some related works and Section 7 concludes.

2 The CSP formalism

A CSP is a quadruple $P = (X, D, C, R)$ where: $X = \{X_1, ..., X_n\}$ is a set of n variables; $D = \{D_1, ..., D_n\}$ is the set of finite discrete domains associated to the CSP variables, D_i includes the set of possible values of the CSP variable X_i; $C = \{C_1, ..., C_m\}$ is a set of m constraints each involving some subsets of the CSP variables. A binary constraint is a constraint which involves two variables; $R = \{R_1, ..., R_m\}$ is a set of relations corresponding to the constrains of C. R_i represents the list of value tuples permitted by the constraint C_i. A CSP P can be represented by a constraint graph $G(X, E)$ where the set of vertices X is the set of the CSP variables and each edge of E connects two variables involved in the same constraint $C_i \in C$.

A binary constraint is called a Not-Equal constraint if it forces the two variables X_i and X_j to take different values (it is denoted by $X_i \neq X_j$). A Not-Equal CSP (NECSP) is a CSP whose all constraints are Not-Equal constraints.

An instantiation $I = (a_1, a_2, ..., a_n)$ is the variable assignment $\{X_1 = a_1, X_2 = a_2, \ldots, X_n = a_n\}$ where each variable X_i is assigned to a value a_i of its domain D_i. A constraint $C_i \in C$ is satisfied by I if the projection of I on the variables involved in C_i is a tuple of R_i. The instantiation I is consistent if it satisfies all the constraints of C, thus I is a solution of the CSP. An instantiation of a subset of the CSP variables is called a partial instantiation. An instantiation is total if it is defined on all the CSP variables. Given a CSP, the main question is to decide its consistency.

Example 2.1 *Take the binary NECSP whose constraint graph is shown in the figure 1. The CSP variables are the vertices $X_1, ..., X_5$ and the domains are include in boxes. Each edge of the constraint graph connecting two verticesX_i and X_j, expresses a Not-Equal constraint between the corresponding CSP variable X_i and X_j.*

3 Dominance in NECSPs

Symmetrical values of a CSP variable are values which have the same semantical relevance to participate in the solutions of the CSP. But, the values of domains in a CSP are not all equally semantical relevant. Some of them can be more likely to participate in solutions than other ones. The values in the first group *dominate* the latter ones. It is important to detect the dominant values in order to consider them prior in assignments. This will reduce the search space. Freuder in [Freuder, 1991] introduced the notion of Substitutability as a weak Interchangeability. In the same spirit, Benhamou in [Benhamou, 1994b] defined a weak symmetry called *Dominance* which extends the Full substitutability notion. Here we adapt the principle of dominance to NECSPs and give some sufficient conditions which leads to a linear algorithm for dominance detection.

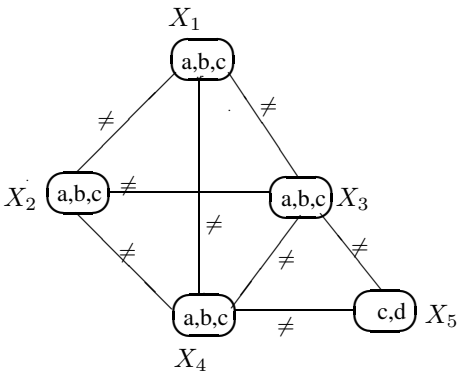


Figure 1: Constraint graph of a NECSP

3.1 The principle of Dominance

Definition 3.1 *[Semantic Dominance] A value a_i dominates another value b_i for a CSP variable $v_i \in V$ (notation $a_i \succeq b_i$) iff [There exist a solution of the CSP which assigns the value b_i to the variable $v_i \Rightarrow$ there exist a solution of the CSP which assigns the value a_i to v_i].*

The value b_i participates in a solution if the value a_i does; otherwise it does not. The value b_i can be removed from D_i without affecting the CSP consistency.

Proposition 3.1 *If $a_i \succeq b_i$ and a_i doesn't participate in any solution of $\mathcal{P}$, then b_i doesn't participate in any solution of $\mathcal{P}$.*

***Proof* 1** *The proof is a direct consequence of Definition 3.1*

Proposition 3.1 allows to remove all the values which are dominated by the value $a_i \in D_i$ without affecting the CSP consistency. Thus, Dominance complements the usual CSP inconsistency methods, which attempt to remove values that doesn't participate in any solution. Dominance can be generalized to values of different variables, but here we restrict the study to values of a same domain.

Remark 3.1
- *Symmetrical domain values are domain values which mutually dominate each other. That is, two values $a_i \in D_i$ and $b_i \in D_i$ are symmetrical iff $a_i \succeq b_i$ and $b_i \succeq a_i$. Thus, a_i participates in a solution of the CSP iff b_i does.*
- *Dominance extends the Full Substitutability notion given in [Freuder, 1991].*
- *Here, Dominance means "more likely to participate in solutions" as it is defined in [Benhamou, 1994b], it*

should not be confused with the Symmetry By Dominance Detection notion (SBDD) [Fahle et al.*, 2001].*

3.2 A sufficient condition for dominance

A symmetry detection algorithm in general discrete finite CSPs is proposed in [Benhamou, 1994a]. Its complexity is exponential in the worst case. It is shown in [Benhamou, 2004] how the symmetry conditions can be simplified in NECSPs and how the symmetrical values can be detected efficiently with a simpler algorithm having a linear time complexity $w.r.t$ to the NECSP size. This result is based on the following property:

Theorem 3.1 *Let a_i and b_i be two values of the domain D_i of a Not-Equals CSP $\mathcal{P}$. If a_i and b_i appear in the same domains of the un-instantiated variables, then they are symmetrical.*

***Proof* 2** *See [Benhamou, 2004].*

It is a very simple property, but very useful for detecting and eliminating symmetrical values of the same domain. By using this property, we can deduce that the values a and b of the domain of the CSP variable X_1 illustrated in Figure 1 are symmetrical. Indeed, they both appear in the domains of X_2, X_3, X_4 and do not appear in the domain of X_5. By a similar reasoning, we can also deduce that the values a and b of the domains of the variables X_2, X_3 and X_4 are symmetrical.

We give in the following the sufficient conditions for dominance which represent the main key of this work.

Theorem 3.2 *Let a_i and b_i be two values of a domain D_i corresponding to a variable X_i of a Not-Equals CSP $\mathcal{P}$, I a partial instantiation of $\mathcal{P}$, and Y the set of un-instantiated variables of $\mathcal{P}$. If the two following conditions:*

1. *$a_i \in D_j \Rightarrow b_i \in D_j$, for all $X_j \in Y$ such that X_j shares a constraint with X_i.*
2. *$a_i \in D_j \Leftrightarrow b_i \in D_j$ for each variable X_j of Y which does not share a constraint with X_i*

hold, then a_i dominates b_i in $\mathcal{P}$.

***Proof* 3** *We have to prove under the conditions (1) and (2) that if b_i participates in a solution, then a_i participates in a solution too. Let $I_{X_i=b_i}$ be a solution of the CSP $\mathcal{P}$ where the variable X_i is instantiated to b_i, we have to show the existence of a solution where the variable X_i is instantiated to a_i. Let $\mathcal{P}'$ be the CSP obtained from $\mathcal{P}$ by removing the value b_i from the domains of the variables having a constraint with X_i where the value a_i does not appear. This operation does not render the domains empty, since $I_{X_i=b_i}$ is a solution of the CSP $\mathcal{P}$. That is, each reduced domain must contain at least two values before removing the value b_i. The CSP $\mathcal{P}'$ is a sub-CSP of $\mathcal{P}$, it has the same set of variables, the same set of constraints, but the reduced domains become sub-sets of the original domains from which they derive.The CSP $\mathcal{P}'$ is more constrained than $\mathcal{P}$. That is, each solution of $\mathcal{P}'$ is a solution of $\mathcal{P}$. According to the conditions (1) and (2) which hold on $\mathcal{P}$, the values a_i and b_i become symmetrical in the CSP $\mathcal{P}'$. The conditions of theorem 3.1 hold, then the value a_i participates in a solution of $\mathcal{P}'$ iff the value b_i does. On the other hand, the CSP $\mathcal{P}_{X_i=b_i}$ resulting from $\mathcal{P}$ by considering the assignment $X_i = b_i$ is identical to the CSP ${\mathcal{P}'}_{X_i=b_i}$ resulting from $\mathcal{P}'$ by considering the assignment $X_i = b_i$. We deduce that $I_{X_i=b_i}$ is a solution of $\mathcal{P}'$, and then there exists a solution $I'_{X_i=a_i}$ of $\mathcal{P}'$ where X_i is instantiated to a_i, since a_i and b_i are symmetrical in $\mathcal{P}'$. As the CSP $\mathcal{P}'$ is more constrained than the CSP $\mathcal{P}$, then $I'_{X_i=a_i}$ is a solution of $\mathcal{P}$. We proved the existence of a solution where a_i is assigned to X_i, thus a_i dominates b_i in $\mathcal{P}$.*

Example 3.1 *Consider the domain of the variable X_3 of Figure 1, the value a dominates the value c.*

3.3 The weakened dominance sufficient conditions

Before introducing the weakened sufficient conditions, we define the notion of assignment trees and failure trees corresponding to the enumerative search method used to prove the consistency of the considered CSP.

Definition 3.2 *We call an assignment tree of a CSP $\mathcal{P}$ corresponding to a given search method and a fixed variable ordering, a tree which gathers the history of all the variable assignments made during its consistency proof, where the nodes represent the variables of the CSP and where the edges out coming from a node X_i are labeled by the different values used to instantiate the corresponding CSP variable X_i.*

The root of the tree is the first variable in the ordering. In this work, the considered backtracking method is Forward Checking [R. M. Haralik and G. L. Elliot, 1980].

In an assignment tree of a CSP, a path connecting the root of the tree to a node defines a partial instantiation of the CSP. The variables of the partial instantiation are the nodes of the considered path. The last node of the path corresponds to the last affected variable in the instantiation or to a variable having an empty domain.

We associate to each inconsistent partial instantiation, corresponding to a given path in the assignment tree, a failure tree defined as follows:

Definition 3.3 *Let T be an assignment tree corresponding to a consistency proof of a CSP $\mathcal{P}$, $I = (a_1, a_2, ..., a_i)$ an inconsistent partial instantiation of the variables X_1,X_2,...,X_i corresponding to the path $\{X_1, X_2, ..., X_i\}$ in T. We call a failure tree of the instantiation I, the sub-tree of T noted by $T_{I=(a_1,a_2,...,a_i)}$ such that:*

1. *The root of the tree T and the root of the sub-tree $T_{I=(a_1,a_2,...,a_i)}$ are joined by the path corresponding to the instantiation I;*
2. *All the CSP variables corresponding to the leaf nodes of $T_{I=(a_1,a_2,...,a_i)}$ have empty domains.*

Example 3.2 *Take the CSP of Figure 1 and apply a forward checking process on it w.r.t the variable ordering $\{X_1, X_2, X_3, X_4, X_5\}$. Figure 2 illustrates the assignment tree of the considered CSP. If we take the partial instantiation $I = (b, a)$ which assigns X_1 to the value b and X_2 to the value a, then the failure tree $T_{I=(b,a)}$ of the instantiation I is shown in the figure 2 (the part in a box).*

We can now give the weakened sufficient conditions of dominance. The main idea is to weaken the dominance conditions of theorem 3.2 when an inconsistent partial instantiation

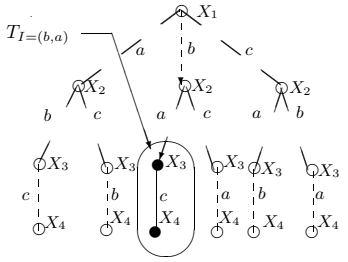


Figure 2: An assignment tree and the failure tree of I=(b,a)

is generated during the search. That is, the conditions of theorem 3.2 are restricted to only the variables involved in the failure tree among the un-instantiated ones.

Theorem 3.3 *Let $\mathcal{P}(X, C, D, R)$ be a CSP, $a_i \in D_i$ and $b_i \in D_i$ two values of the domain D_i of the current CSP variable X_i under instantiation, $I_0 = (a_1, ..., a_{i-1})$ a partial instantiation of the $i-1$ variables instantiated before X_i such that the extension $I = I_0 \bigcup \{a_i\} = (a_1, ..., a_{i-1}, a_i)$ is inconsistent, $T_{I=(a_1,...,a_{i-1},a_i)}$ is the failure tree of I and $Var(T_{I=(a_1,...,a_{i-1},a_i)})$ the set of variables corresponding to the nodes of $T_{I=(a_1,...,a_{i-1},a_i)}$. If the two following conditions:*

1. *$b_i \in D_j \Rightarrow a_i \in D_j$, for all $X_j \in Var(T_{I=(a_1,...,a_{i-1},a_i)})$ such that X_j shares a constraint with X_i.*
2. *$a_i \in D_j \Leftrightarrow b_j \in D_i$ for each other variable X_j of $Var(T_{I=(a_1,...,a_{i-1},a_i)})$ which do not share a constraint with X_i*

hold, then the extension $J = I_0 \bigcup \{b_i\} = (a_1, ..., a_{i-1}, b_i)$ is inconsistent.

***Proof* 4** *Let $\mathcal{P}'(V', C', D', R')$ be a sub-CSP of the CSP $\mathcal{P}(V, C, D, R)$ such that $V' = Var(T_{I=(a_1,...,a_{i-1},a_i)}) \cup X_i$ and $C' \subseteq C$, $D' \subseteq D$ and $R' \subseteq R$ are the restrictions of C,D, and R to the variables of V'. By the hypothesis, $T_{I=(a_1,...,a_{i-1},a_i)}$ is a failure tree of I in $\mathcal{P}$. This implies that the assignment of X_i to the value a_i leads to a failure in $\mathcal{P}'$. In other words, a_i does not participate in any solution of $\mathcal{P}'$. By the hypothesis, the values a_i and b_i verify the condition of theorem 3.2 when restricted to the variables of $Var(T_{I=(a_1,...,a_{i-1},a_i)})$. This means that a_i dominates b_i in the CSP $\mathcal{P}'$. By application of proposition 3.1, we deduce that the value b_i does not participate in any solution of $\mathcal{P}'$. This implies that the partial instantiation $J = I_0 \bigcup \{b_i\} = (a_1, ..., a_{i-1}, b_i)$ is inconsistent in $\mathcal{P}$. (QED)*

This previous property of dominance is a weakening of the conditions of Theorem 3.2 when the current partial instantiation leads to an inconsistency. The case of partial inconsistent instantiation is important, because it allows to prune the consistency proof tree of a CSP $w.r.t$ Proposition 3.1.

Some dominances not captured by Theorem 3.2 can result from these weakened conditions. Let us consider for instance the CSP of Figure 1. If we take the inconsistent partial instantiation $I = (b, a)$ of the variables X_1 and X_2, then the two values a and c of the domain of the current variable X_2 dominate each other (symmetrical) by application of Theorem 3.3, whereas the conditions of both theorems 3.1 and 3.2 are not verified. The branch corresponding to the assignment of X_2 to c is not explored in the consistency proof tree thanks to Theorem 3.3. This defines a dominance cut which we use in Section 4 to shorten the CSP search tree.

3.4 Dominance detection

Now we deal with the dominance detection problem. Dominance detection is based on the conditions of theorem 3.3. The algorithm sketched in Figure 3 computes the values dominated by a value a_i of a given domain D_i $w.r.t$ the conditions of theorem 3.3. These values form the class of dominance of a_i which we denote by cl(a_i)).

procedure $weak\_dominance(a_i \in D_i, Var(T_{I=(a_1,...,a_i)}))$,
var cl(a_i):class);
input: a value $a_i \in D_i$, a set of variables $Var(T_{I=(a_1,...,a_i)})$
Output: the class $cl(a_i)$ of the dominated values by a_i.
begin
  cl(a_i):=$\{a_i\}$
  for each $d_i \in D_i$-$\{a_i\}$ **do**
    for each domain D_k of variables
    of $Var(T_{I=(a_1,...,a_i)})$
      if ($c_{ik} \in C$ **and** ($a_i \in D_k \Rightarrow d_i \in D_k$))
      or
      ($c_{ik} \notin C$ **and** ($a_i \in D_k \Leftrightarrow d_i \in D_k$))
      then cl(a_i):=cl(a_i)$\cup\{d_i\}$
end

Figure 3: The algorithm of dominance search in NECSPs

Complexity: Let n be the number of variables of the NECSP, and d the size of the largest domain. It is easy to see that the algorithm of Figure 3 can run at most d times the first loop and at most n times the second one. It then computes the class $cl(d_i)$ of dominated values with a complexity $\mathcal{O}(nd)$ in the worst case. This algorithm has a linear complexity $w.r.t$ the NECSP size.

In theory, this algorithm has the same complexity as the one of the algorithm described in [Benhamou, 2004] in the worst case. But, this new algorithm detects dominance rather than only symmetry and detect some symmetries which are not detected by the algorithm in [Benhamou, 2004]. All the symmetries detected in [Benhamou, 2004] are detected with this new algorithm, since it works on a weakened dominance condition. That is, the dominance condition is verified on a reduced subset of the non-instantiated variables (the ones of $Var(T_{I=(a_1,...,a_i)})$) rather on the hole set of un-instantiated variable as it is done in [Benhamou, 2004].

4 Exploiting Dominance in NECSPs

Now, we show how the dominance property given in Theorem 3.3 is exploited to increase the efficiency of NECSP backtracking algorithms. This property can be exploited in all enumerative resolution methods. Here we implemented a *Simplified Forward Checking* method (denoted by SFC) adapted to NECSPs which we want improve by adding the dominance property.

The principle of the Forward Checking [R. M. Haralik and G. L. Elliot, 1980] is based on filtering the domains of the non-instantiated variables $w.r.t$ the instantiated one.

In the case of NECSPs, the filtering is simplified. It consists only in removing the value d_i from the domains of the future variables having a constraint with the current variable v_i under instantiation. This results in a *Simplified Forward Checking* which we considered in our implementation. The rest of the method is just the classic backtracking.

```
Procedure SFC-weak-dom(D, I, i, VFT : a list);
input: a set of domains D, I = (d_1, ..., d_i) a partial instantiation
of variables {v_1, ..., v_i}; i the index of the current variable and VFT
the set of variables Var(T_I) of the failure tree T_I (at the beginning VFT is empty).
var empty:boolean;
var VFT_tmp: a list;
var VFT_old: a list;
begin
    if i = n then [d_1, d_2, ..., d_i] is a solution, print(I), stop
    else
    begin
        empty:=false;
        VFT_tmp:=VFT;
        VFT_old:=VFT;
        for each v_j ∈ V, such as C_ij ∈ C, v_j ∈ future(v_i) do
            if not(empty) and d_i ∈ D_j then
                begin
                    D_j=D_j-{d_i};
                    if D_j=∅ then
                    begin
                        undo filtering effects;
                        add(v_j, VFT);
                        empty:=true;
                    end
                end
        if not(empty) then
        begin
            v_{i+1}=next-variable(v_i)
            repeat
                take d_{i+1} ∈ D_{i+1}
                D_{i+1}=D_{i+1} − d_{i+1}
                I = [d_1, d_2, ..., d_i, d_{i+1}];
                VFT_tmp:=VFT∪ VFT_tmp;
                VFT:=VFT_old;
                SFC-weak-dom(D, I, i + 1, VFT);
                weak_dominance(d_{i+1} ∈ D_{i+1},VFT,Cl(d_{i+1}));
                D_{i+1}=D_{i+1} − Cl(d_{i+1});
            until D_{i+1} = ∅
        end
        VFT:=VFT_tmp;
        add(v_i, VFT);
    end
end
```

Figure 4: The SFC method augmented by dominance

Theorem 3.3 allows to prune k-1 branches in the search tree if there are k dominated values by a dominant value which is shown to not participating in any solution. If $Cl(d_i)$ denote the class of values of the domain D_i which are dominated by d_i, then we consider only the value d_i, since the other values of $Cl(d_i)$ are redundant.

Figure 4 sketches the SFC procedure augmented by the dominance property of theorem 3.3 (notation SFC-weak-dom)) which decides just the consistency of a NECSP. This method can be easily modified to compute all non-symmetrical solutions of a NECSPs. The structure $future(v_i)$ encodes the set of non-instantiated variables remaining after the instantiation of v_i, and $next\text{-}variable$ a function which encodes the (DomDeg) heuristic. It consists in minimizing the ratio

$$r = \frac{\mid D_j \mid}{Degree(v_j)}$$

where $Degree(v_j)$ denotes the number of constraints of the initial CSP in which the variable v_j is involved to select the next variable. In the sequel SFC will denote the SFC method augmented by the (DomDeg) heuristic.

5 Experiments

We will now evaluate the performances of our implementation. The tests are made on both randomly generated graph coloring instances and some graph coloring benchmarks of the 2^{nd} challenge of Dimacs (http://dimacs.rutgers.edu/Challenges). Graph coloring is trivially expressed as a NECSP. We will test and compare the Simplified Forward Checking augmented by the symmetry property defined in [Benhamou, 2004] (SFC-sym), the Simplified Forward Checking augmented by the advantage of the dominance property of theorem 3.3 (SFC-weak-dom) and an improved version [Sewell, 1995] of the well known method DSATUR [D. Brelaz, 1979]. This method is based on a heuristic which consists in coloring the vertices of a graph according to their *saturation degree*. The saturation degree of a vertex is the the number of different colors to which it is adjacent. The DSATUR heuristic repeatedly chooses a vertex having a maximal saturation degree and colors it with the lowest-numbered color possible. The complexity indicators are the number of nodes and CPU time. The source code is written in C and compiled on a P4 2.8 GHz - RAM 1 Go.

5.1 Random graph coloring problems

Random graph coloring problems are generated according to the parameters:(1) n the number of vertices (the variables), (2) Cls the number of colors (the domain values) and (3) d the density which is the ratio expressing the number of constraints to the total number of possible constraints.
For each test corresponding to some fixed value of the parameters n, Cls and d, a sample of 100 instances are randomly generated and the measures (CPU time, nodes) are taken in average.

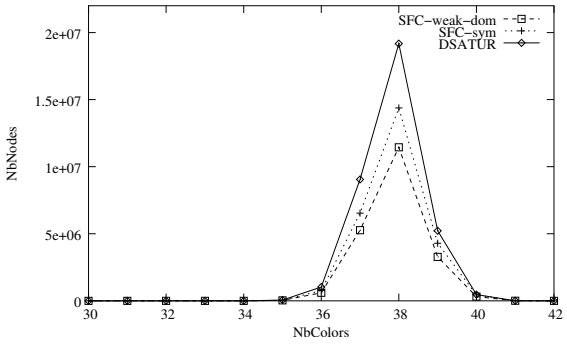


Figure 5: The curves representing the number of nodes

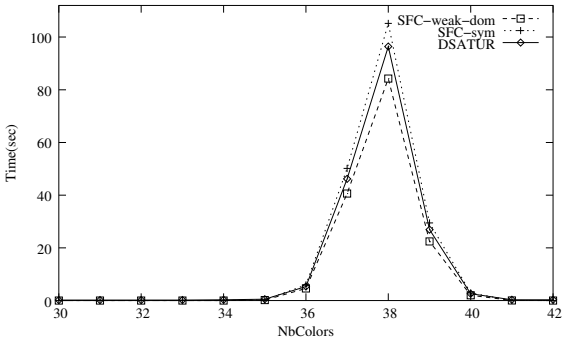


Figure 6: The curves representing the CPU times

Figures 5 and 6 give the performances of the methods DSATUR, SFC-sym and SFC-weak-dom in number of nodes of the search tree, respectively, in CPU time (in seconds) on random graph coloring problems where the number of variables is fixed to $n = 100$ and the density to $d = 0.9$. SFC without symmetry was not able to solve these instances in reasonable time. This is why we did not give its curve. We can see that SFC-weak-dom, generates less nodes than both DSATUR and SFC-sym, and spends less time than both methods to solve the problems. This proves that SFC-weak-dom detects and eliminates more symmetries than the other methods and the detection is faster in SFC-weak-dom. The symmetry behavior is now shown. DSATUR eliminates only the global[1] symmetry between colors, then generate more nodes than FC-sym. SFC-sym detects more symmetries than DSATUR since it detects both the global symmetry and local [2] symmetry, but less than SFC-dom-weak which considers dominance. Only SFC-weak-dom captures the dominance, and the detection is faster than in SFC-sym, thanks to the weakened condition. These remarks are confirmed by the following Dimacs benchmarks where the gain is important.

5.2 Dimacs graph coloring benchmarks

Table 1 shows the results of the methods on some graph coloring benchmarks of Dimacs. It gives the number of nodes of the search tree and the CPU time for each method. We seek for each of them the minimal number k of colors needed to color the vertices of the corresponding graph (the chromatic number). The search of the chromatic number consist in proving the consistency of the problem with k colors (existence of a k-coloration of the graph); and in proving its inconsistency when using $k-1$ colors. The symbol "-" means that the corresponding method does answer the question in one hour.

We can remark that only SFC-weak-dom is able to solve the known hard problem "DSJR500.1c" which, as far as, we know it has never been solved by an exact method. We can see that SFC-dom-weak outperforms both DSATUR and SFC-sym, and SFC-sym is better than DSATUR on these problems. DSATUR is the less effective since it does not succeed to solve 9 benchmarks among the 22 proposed. For space reason, we report here the results on the most relevant Dimacs problems to compare DSATUR and our method, but it is important to inform the reader that all the others DIMACS problems which are solved by DSATUR are also solved by SFC-weak-dom with a comparable performance.

[1]The trivial symmetry of the initial problem which consists in the interchangeable colors.

[2]The existing symmetry between values at each node of the search tree.

Pb *instances* $(V-E)$	k	DSATUR		SFC-SYM		SFC-dom-weak	
		N	T	N	T	N	T
queen8_8 (64-728)	9	1581661	4.3	1368441	6.1	1353680	6.1
queen8_12 (96-1368)	12	162	0.0	460	0.0	460	0.0
myciel5 (47-236)	6	378310	0.6	72966	0.2	21278	0.0
myciel6 (95-755)	7	-	-	83157279	556.0	29754513	190.2
le450_5a (450-5714)	5	-	-	1408	0.1	1395	0.1
le450_5b (450-5734)	5	-	-	19884	0.6	19763	0.5
le450_25a (450-8260)	25	425	0.1	450	0.1	450	0.1
le450_25b (450-8263)	25	425	0.0	450	0.1	450	0.1
1-FullIns_3 (30-100)	4	37	0.0	51	0.0	50	0.0
1-FullIns_4 (93-593)	5	-	-	8885	0.0	1368	0.0
2-FullIns_3 (52-201)	5	156663424	193.6	678	0.0	359	0.0
qg.order30 (900-26100)	30	1680	0.2	1169	0.2	1162	0.2
qg.order40 (1600-62400)	40	-	-	12089785	302.0	10814593	266.6
school.1 (385-19095)	14	371	0.2	568	0.4	555	0.4
school_nsh (352-14612)	14	338	0.2	352	0.4	352	0.4
wap05a (905-43081)	50	855	1.3	905	1.4	905	1.4
mug88_25 (88-146)	4	-	-	22643	0.0	1631	0.0
mug100_25 (100-166)	4	-	-	99917	0.2	515	0.0
ash608GPIA (1216-7844)	4	10242	0.5	1742	0.5	1707	0.5
ash958GPIA (1916-12506)	4	-	-	10252	2.2	7167	1.6
R125.5 (125-3838)	36	1357573	9.1	55952	0.4	1051	0.0
DSJR500.1c (500-121275)	84	-	-	-	-	28044984	3096.0

Table 1: Dimacs graph coloring benchmarks

6 Some related works

- In [P. Van Hentenryck *et al.*, 2003] authors studied three classes of CSPs where symmetry is tractable. The value-Interchangeable CSPs (ICSPs) class is in relation with our work. That is, when all the variable domains of a NECSP are the same (as in the graph coloring problem), it becomes an ICSP. For this particular case, the value symmetry elimination technique used in [P. Van Hentenryck *et al.*, 2003] for the ICSP class seems to be equivalent to the global symmetry elimination described in [Benhamou, 2004]. But,in general NECSPs are not ICSPs and the dominance/symmetry detected by the procedure of figure 3 are not considered in the ICSP symmetry breaking.
- On other hand Gent introduced in [I. Gent, 2001] a symmetry constraint to eliminate what he calls indistinguishable values. His approach works by addition of symmetry constraints rather than dynamic detection of symme-

try. This technique may be used to break some of the trivial global symmetry such as the one of the graph coloring problem for example. However, it does not deal with the local symmetry or the dominance which we detect by using the dominance procedure of figure 3.

- In [A. Ramani *et al.*, 2004], authors investigated a static approach to break symmetry in exact graph coloring problem reduced to a hybrid constraint representation formed by: boolean constraints and pseudo boolean constraints resulting from a 0-1 ILP reduction of the problem. Two kinds of symmetry are exploited: the instance-dependent symmetry and the instance-independent symmetry. Our approach is dynamic whereas theirs is static and does not deal with dominance. These are two different approaches which can be combined. Our method seems to be more efficient for solving graph coloring problems.
- The GE-tree method [Colva M. Roney-Dougal *et al.*, 2004] breaks all value symmetries at each node of the search tree of a CSP. This method can eventually be used to break the symmetries we are dealing with in NECSPs during search, but it will be time consuming, since its complexity at each node of the search tree, is approximately $O(n^4d^4)$ in the worst case. Our method detects the needed class of symmetry of a value in $O(nd)$ and detects dominance which is not considered by the GE-tree technique.
- Recently in [Puget, 2005b; 2004], Puget studied symmetry between variables involved in a global AllDiff constraint. This is a particular case of NECSP, since the Alldiff constraint can be expressed as a NECSP whose constraint graph is complete. The symmetry elimination technique is static, it consists in adding some extra constraints to the problem formulation to eliminate the symmetries between variables. The most important result here is the fact that only a linear number of extra constraints is needed to break all the variable global symmetries. Our approach is dynamic, it focuses on local dominance/symmetry between values. Our method can be safely combined with Puget's one. It will be interesting to study the behavior of the resulting method on complete NECSPs (Alldiff constraints).

7 Conclusion

In this work we extended the symmetry principle to dominance and weakened the symmetry/dominance sufficient conditions in Not-Equals constraint networks when an inconsistent partial instantiation is generated. We implemented a more efficient dominance search algorithm which detects both symmetry and which captures the dominance. We exploited the new dominance property in a Simplified Forward Checking backtracking algorithm adapted to NECSPs. Experiments are carried on both random generated graph coloring problems and Dimacs graph coloring benchmarks. The obtained results show that reasoning by dominance is profitable for solving NECSPs. Our method beats the well known exact method for solving graph coloring.

Further investigation consists first in extending the dominance property to values of different domains, then investigate dominance detection in more general CSPs. Another point is to combine some CSP decomposition methods with our method to improve Not-Equals constraint networks solving.

References

[A. Ramani *et al.*, 2004] A. Ramani, F.A. Aloul, I.L. Markov, and K.A. Sakallah. Breaking instance-independent symmetries in exact graph coloring. *In DATE*, pages 324–329, 2004.

[Benhamou and Sais, 1992] B. Benhamou and L. Sais. Theoretical study of symmetries in propositional calculus and application. *CADE-11, Saratoga Springs,NY, USA*, 1992.

[Benhamou, 1994a] B. Benhamou. Study of symmetry in constraint satisfaction problems. *In PPCP'94*, 1994.

[Benhamou, 1994b] B. Benhamou. Theoretical study of dominance in constraint satisfaction problems. *6th International Conference on Artificial Intelligence: Methodology, Systems and Applications (AIMSA-94), Sofia, Bulgaria, september*, pages 91–97, 1994.

[Benhamou, 2004] B. Benhamou. Symmetry in not-equals binary constraint networks. *SymCon'04 : 4th International Workshop on Symmetry and Constraint Satisfaction Problems*, 2004.

[Cohen *et al.*, 2005] D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. *In proceedings of CP*, pages 17–31, 2005.

[Colva M. Roney-Dougal *et al.*, 2004] Colva M. Roney-Dougal, Ian P. Gent, Tom Kelsey, and Steve A. Linton. Tractable symmetry breaking using restricted search trees. *In proceedings of ECAI-04*, 2004.

[D. Brelaz, 1979] D. Brelaz. New methods to color the vertices of a graph. *In the communications of the ACM 22*, pages 251–256, 1979.

[F. Focacci and M. Milano, 2001] F. Focacci and M. Milano. Global cut framework for removing symmetries. In *CP'01*, volume 2239 of *LNCS*, pages 77–82. Springer Verlag, 2001.

[F.A. Aloul *et al.*, 2003] F.A. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah. Solving difficult sat instances in the presence of symmetry. *In IEEE Transaction on CAD, vol. 22(9)*, pages 1117–1137, 2003.

[F.A. Aloul *et al.*, 2004] F.A. Aloul, A. Ramani, I.L. Markov, and K.A. Sakallah. Symmetry breaking for pseudo-boolean satisfiabilty. *In ASPDAC'04*, pages 884–887, 2004.

[Fahle *et al.*, 2001] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *International conference on constraint programming*, volume 2239 of *LNCS*, pages 93–108. Springer Verlag, 2001.

[Freuder, 1991] E.C. Freuder. Eliminating interchangeable values in constraints satisfaction problems. *Proc AAAI-91*, pages 227–233, 1991.

[I. Gent, 2001] I. Gent. A symmetry breaking constraint for indistinguishable values. In *SymCon'01*, 2001.

[I.P. Gent *et al.*, 2002] I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In *CP'02*, volume 2470 of *LNCS*, pages 415–430. Springer Verlag, 2002.

[James Crawford *et al.*, 1996] James Crawford, Matthew L. Ginsberg, Eugene Luck, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *KR'96*, pages 148–159. Morgan Kaufmann, San Francisco, California, 1996.

[Jean F. Puget, 2002] Jean F. Puget. Symmetry breaking revisited. In *CP'02*, volume 2470 of *LNCS*, pages 446–461. Springer Verlag, 2002.

[Krishnamurty, 1985] B. Krishnamurty. Short proofs for tricky formulas. *Acta informatica*, (22):253–275, 1985.

[M.R. Garey and D.S. Johnson, 1979] M.R. Garey and D.S. Johnson. Computers and intractability: A guide to the theory of np-completeness, w.h. freeman. 1979.

[P. Van Hentenryck *et al.*, 2003] P. Van Hentenryck, P. Flener, J. Pearson, and M. Argen. Tractable symmetry breaking for csps with interchangeable values. In *IJCAI*, pages 277–282, 2003.

[Puget, 1993] Jean F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *ISMIS*, 1993.

[Puget, 2004] Jean F. Puget. Breaking symmetries in all different problems. *SymCon'04 : 4th International Workshop on Symmetry and Constraint Satisfaction Problems*, 2004.

[Puget, 2005a] Jean F. Puget. Breaking all value symmetries in surjection problems. *In proceedings of CP*, pages 490–504, 2005.

[Puget, 2005b] Jean F. Puget. Breaking symmetries in all different problems. *In proceedings of IJCAI*, pages 272–277, 2005.

[R. M. Haralik and G. L. Elliot, 1980] R. M. Haralik and G. L. Elliot. Increasing tree search efficiency for constraint satisfaction problems. *Artificial Intelligence 14*, pages 263–313, 1980.

[Sewell, 1995] Edward C. Sewell. An improved algorithm for exact graph coloring. *DIMACS series on Discrete Mathematics and Theorical Computer Science*, 1995.

GAPLex: Generalised Static Symmetry Breaking

Chris Jefferson, Tom Kelsey, Steve Linton and **Karen Petrie**
chris.jefferson@comlab.ox.ac.uk, {tom,sal,kep}@dcs.st-and.ac.uk

Abstract

We describe a novel algorithm that statically breaks symmetry in CSPs by using computational group theory during search. This algorithm extends and generalises the commonly used "double lex" method for breaking symmetry in matrices. We show that our new symmetry breaking method, GAPLex, is sound (will neither lose solutions nor return incorrect solutions) and complete (will return exactly one member from each class of symmetrically equivalent solutions). We demonstrate that our implementation of GAPLex is competitive with other methods, being effectively applicable to CSPs with large domains and less than full variable and/or value symmetry. We also describe how GAPLex can be combined with incomplete symmetry breaking methods – such as double-lex – to provide fast and complete symmetry breaking. We believe this to be the first method that successfully combines the posting of symmetry breaking constraints before search, with symmetry breaking by analysis of search states.

1 Introduction

Constraint satisfaction problems (CSPs) are often highly symmetric. Given any solution, there are others with are equivalent in terms of the underlying problem. Symmetries may be inherent in the problem, or be created in the process of representing the problem as a CSP. Without symmetry breaking (henceforth SB), many symmetrically equivalent solutions may be found and, in some ways more importantly, many symmetric equivalent parts of the search will be explored. A SB method aims to avoid both these problem.

Most SB algorithms fall into one of two groups. The first, known as static SB algorithms, decide before search which assignments will be permitted and which will be forbidden and are therefore independent of search ordering. The second group, known as dynamic symmetry breaking methods instead choose which assignment will be permitted during search.

Existing static symmetry breaking methods work by adding extra constraints to the CSP, either before or during search. This typically involves constraints that rule out solutions by enforcing some lexicographic-ordering on the variables of the problem [Crawford *et al.*, 1996; Frisch *et al.*, 2002]. STAB [Puget, 2003] is a SB algorithm for solving BIBDs which avoids adding an exponential number of constraints at the start of search by adding constraints during search on the row of the BIBD currently being assigned.

Dynamic symmetry breaking methods operate in a number of ways, including posting constraints that rule out search at states that are symmetrically equivalent to the current assignment [Backofen and Will, 1999; Gent and Smith, 2000], building the search tree such that symmetric nodes are avoided [Roney-Dougal *et al.*, 2004] or backtracking from nodes that are symmetrically equivalent to root nodes of subtrees that have previously been fully explored [Fahle *et al.*, 2001; Focacci and Milano, 2001; Puget, 2003]. These methods, especially in the case that symmetries are represented by permutation groups, are related to a class of algorithms first described in [Brown *et al.*, 1988].

The major weakness of static SB methods, compared to dynamic methods, is that they can increase the number of search nodes visited [Gent *et al.*, 2002], as the first solution which would have been found is forbidden by the symmetry breaking constraints. The major advantage of static symmetry breaking methods is that adhoc problem specific simplifications often perform very well and powerful implied constraints can be derived from these constraints [Frisch *et al.*, 2004]. Moreover, since static SB does not depend on previously encountered search nodes, it can be used when searching in parallel on multiple machines.

Other methods of breaking symmetry do exist, for example a CSP can be reformulation of the CSP so that either the number of symmetries is reduced, or some other approach can be applied more effectively [Kelsey *et al.*, 2004; Meseguer and Torras, 2001]. However, these are currently problem specific and not discussed here.

Groups are the mathematical structures that best encapsulate symmetry. Many powerful algorithms for in-

vestigating group-theoretic questions are known, and have been efficiently implemented in systems such as GAP [GAP, 2000] and MAGMA [Bosma and Cannon, 1993]. Previously, computational group theory (henceforth CGT) has been used with a high level of success in dynamic symmetry breaking with both Symmetry Breaking During Search (SBDS) [Gent *et al.*, 2002] and Symmetry Breaking via Dominance Detection (SBDD) [Gent *et al.*, 2003], allowing generic systems which can handle over 10^{25} symmetries. Recent advances [Linton, 2004] have provided the CGT tools necessary to efficiently implement static symmetry breaking.

In this paper we address the open research question: can we implement lex-ordering with a CGT approach in such a way that we retain the best features of static symmetry breaking while gaining the speed and flexibility of a general group-theoretic framework? Our contribution is twofold. We first describe a novel SB method, GAPLex, which involves both ordering constraints and symmetry information regarding the current state of search to break as many solution symmetries of a CSP as are required. We also demonstrate that GAPLex can be combined with previous fast but incomplete static lex-orderings to provide fast and complete SB.

In the remainder of this introduction we give a detailed backgound of SB by lex-ordering, a basic description of permutation groups acting on literals of CSPs, and describe existing approaches to the use of CGT to break symmetries dynamically. In Section 2 we motivate and describe GAPLex, and provide empirical results for our implementation. Section 3 describes our combination of GAPLex with static lex-orderings, again with empirical results. We conclude with a summary of our results and an outline of future research in this area.

1.1 Lex-ordering to break symmetries

Puget [Puget, 1993] proved that whenever a CSP has symmetry, it is possible to find a 'reduced form', with the symmetries eliminated, by adding constraints to the original problem and showed such a form for three CSPs. Following this, the key advance was to show a method whereby such a set of constraints could be generated. Crawford, Ginsberg, Luks and Roy showed a general technique, called "lex-leader", for generating such constraints for any variable symmetry [Crawford *et al.*, 1996].

The idea behind lex-leader is essentially simple. For each equivalence class of assignments under our symmetry group, we choose one to be canonical. We then add constraints before search starts which are satisfied by canonical assignments and not by any others. We generate canonical assignments by choosing an ordering of the variables and representing assignments as tuples under this variable ordering. Any permutation of variables g maps tuples to tuples, and the lexicographically least of these is our canonical assignment. This gives the set of constraints

$$\forall g \in G,\ V \preceq_{\text{lex}} V^g$$

where V is the vector of the variables of the CSP, $\preceq_{\text{lex}}$ is the standard lexicographic ordering relation, defined by $AD \preceq_{\text{lex}} BC$ iff either $A < B$ or $A = B$ and $D \leq C$, and V^g denotes the permutation of the variables by application of the group element.

1.2 Group theory for CSPs

Definition 1 *A CSP L is a set of constraints $\mathcal{C}$ acting on a finite set of variables $\Delta := \{A_1, A_2, \ldots, A_n\}$, each of which has finite domain of possible values $D_i := D(A_i) \subseteq \Lambda$. A* solution *to L is an instantiation of all of the variables in Δ such that all of the constraints in $\mathcal{C}$ are satisfied.*

Constraint logic programming systems typically model CSPs using constraints over finite domains. The usual search method is depth-first, with values assigned to variables at choice points. After each assignment a partial consistency test is applied: domain values that are found to be inconsistent are deleted, so that a smaller search tree is produced.

Statements of the form $(Var = val)$ are called *literals*, so a partial assignment is a conjunction of literals. We denote the set of all literals by χ, and denote variables by Roman capitals and values by lower case Greek letters.

Definition 2 *Given a CSP L, with a set of constraints $\mathcal{C}$, and a set of literals χ, a* symmetry *of L is a bijection $f : \chi \to \chi$ such that a full assignment A of L satisfies all constraints in $\mathcal{C}$ if and only if $f(A)$ does.*

We denote the image of a literal $(X = \alpha)$ under a symmetry g by $(X = \alpha)g$. The set of all symmetries of a CSP form a *group*: that is, they are a collection of bijections from the set of all literals to itself that is closed under composition of mappings and under inversion. We denote the symmetry group of a CSP by G.

Definition 3 *Let G be a group of symmetries of a CSP. The* stabiliser *of a literal $(X = \alpha)$ is the set of all symmetries in G that map $(X = \alpha)$ to itself. This set is itself a group. The* orbit *of a literal $(X = \alpha)$, denoted $(X = \alpha)^G$, is the set of all literals that can be mapped to $(X = \alpha)$ by a symmetry in G. The orbit of a node is defined similarly.*

Given a collection $\mathcal{S}$ of literals, the *pointwise* stabiliser of $\mathcal{S}$ is the subgroup of G which stabilises each element of $\mathcal{S}$ individually. The *setwise* stabiliser of $\mathcal{S}$ is the subgroup of G that consists of symmetries mapping the set $\mathcal{S}$ to itself.

1.3 Using GAP to break CSP symmetries

There have been three successful implementations of SB methods which use GAP to provide answers to symmetry related questions during search. All three combined GAP with the constraint solver ECL^{i}PS^e [Wallace *et al.*, 1997]. GAP-SBDS [Gent *et al.*, 2002] is an implementation of symmetry breaking during search: at each search node, constraints are posted which ensure that no symmetrically equivalent node will be visited later in search. Enough pruning of the search tree

is made, in general, to make GAP-SBDS more efficient than straightforward search. The number of SB constraints is linear in the size of the group, making GAP-SBDS unattractive for groups of size greater than about 10^9.

GAP-SBDD [Gent *et al.*, 2003] uses GAP to check that the next assignment is not equivalent to a state which is the root of a previously explored sub-tree. Again, the overhead of finding (or failing to find) these group elements is usually more than offset by the reduction in search due to early backtracking. Larger groups – up to about 10^{25} – can be dealt with, simply because the answer from GAP is a straight yes or no to the dominance question; the overhead of passing constraint information is not present. GAP also reports literals that can be safely deleted because setting them would have lead to dominance. Provided that the cost of computing these safe deletions is low enough, the domain reductions are a gain over not making them. We follow the same idea in this paper; we search for literals that, if set, would have lead to a non-lex-least assignment.

The third use of GAP is the building of search trees that, by construction, have no symmetrically equivalent nodes and contain a member from each solution equivalence class: GE-trees [Kelsey *et al.*, 2004].

In this paper we aim to build upon the strengths of these existing frameworks by using lex-ordering as the main SB technique, using GAP to decide if the current partial assignment is lex-smallest of the orbit of the assignment under the symmetry group.

2 GAPLex

2.1 Motivation and rationale

Both lex-ordering and CGT-based SB methods are effective and attractive options for breaking symmetries in CSPs. Our aim is to implement lexicographic static symmetry breaking using the CGT methods used in GAP-SBDS and GAP-SBDD, thus combining useful features of both approaches.

We want to enforce a lex ordering on the literals of a CSP so that only solutions that are minimal in the ordering are returned after complete backtrack search with propagation. Moreover, we want this to work with any symmetry structure induced by the formulation of the CSP.

The key idea is that we can compute the minimum image, under a symmetry group G, of those literals that represent the ground variables in any branch of the search tree. By minimum image, we mean the lex-least ordered list of literals that can be obtained by applying group elements (symmetries) to our set of ground literals. If the minimum image of our current partial assignment is lex-smaller than that assignment, then it is safe to backtrack from the current search node: further search will either fail to find a solution or return a solution that is not the lex-smallest in its equivalence class.

Recent advances in algorithms for finding minimal images have lead to dramatic improvements in both worst-case and apparent average-case time complexities [Linton, 2004]. We use these more efficient CGT methods to obtain the minimal images and decide the ordering predicate. As a useful extension to the main technique, we can also use CGT to identify those literals involving non-ground variables that, if taken as assignments, would result in failure of the lex-smaller test. The assignments of any such literals can be ruled out immediately by deleting the values from their respective domains. These domain deletions, together with the propagation of the domain deletion decisions, reduce the search required to find solutions (or confirm that no solutions exist) for the CSP.

2.2 Motivating example

Suppose that we wish to solve

$$\frac{A}{10B+C}+\frac{D}{10E+F}+\frac{G}{10H+I}=1$$

where each variable takes a value from 1 to 9. There are 3! permutations of the summands which preserve solutions. Suppose now that during search for all solutions we have made the partial assignment $PA : A = 2$, $B = 3$, $C = 8$, $D = 2$, $E = 1$. One symmetry maps this to $G = 2$, $H = 3$, $I = 8$, $A = 2$, $B = 1$; or, in sorted order, $A = 2$, $B = 1$, $G = 2$, $H = 3$, $I = 8$.

Under the variable ordering $ABCDEFGHI$, this is lexicographically smaller than PA (since $B = 1$ is smaller than $B = 3$) so we would backtrack from this position. Notice that although we map a state which includes $A = 2$ into a smaller state we *can't* do it by mapping the literal $A = 2$ to itself.

Even if we can't immediately backtrack, there are often safe domain deletions that can be made. For example if we have only assigned $A = 7$, it is safe to remove values 1 through 6 from the domains of D and G, since making any of these assignments would lead to an immediate backtrack.

2.3 The GAPLex algorithms

We have a CSP and a symmetry group, G, for the CSP. G acts on the set of literals (variable-value pairs) of the problem, written as an initial subset of the natural numbers. The set of literals forms a variables $\times$ values array, so that the literals of the lex-least variable are $1, 2, \ldots, |dom(V_1)|$, etc. The GAPLex method, applied at a node N in search, proceeds as follows :

Require: $PA \leftarrow$ current partial assignment
Require: $Var \leftarrow$ next variable w.r.t. any fixed choice heuristic
Require: $val \leftarrow$ next value w.r.t. lex-least value ordering
1: set $Var = val$ and propagate
2: add $Var = val$ to PA
3: $T \leftarrow$ literals involving unassigned variables
4: pass PA and T to the GAPLex CGT test
5: **if** the test returns **false then**
6: backtrack
7: **else**
8: the test returns **true** and a list of literals, D
9: **for** $(X = \alpha) \in D$ **do**

10: remove α from the domain of X and propagate
11: **end for**
12: continue search
13: **end if**
14: **if** $Var = val$ does not lead to a solution **then**
15: set $Var \neq val$ and propagate
16: **if** a solution is obtained **then**
17: check that the solution is not isomorphic to any previous solution
18: **else**
19: move to next search node
20: **end if**
21: **end if**

There are several points of interest. The list T passed to the CGT test contains only those literals that involve the current domains of non-ground variables, as opposed to the domains before search. In this sense T is the smallest list we can pass, making the CGT test as efficient as possible. The method – if applied at every node in search – is sound, since we only backtrack away from solutions that are not lex-least, and we make no domain deletions involving lex-least solutions. This method is very much in the spirit of GAP-SBDD; the aim is to replace dominance detection by the power and simplicity of lex-ordering SB heuristics. Another way of looking at the method is as a propagator for (unposted) lex-ordering SB constraints. The method backtracks and reduces domains in line with the constraints that a static lex-ordering would have posted before search.

As in other CGT-based SB methods, we can only expect a win if the cost of the CGT test is less than the cost of performing the search needed without early backtracking and early domain deletions provided by the test. By using highly efficient implementations of powerful permutation group algorithms, we can achieve this goal. The GAPLex CGT test, written in GAP, is specified as follows:

Require: G – a symmetry group for a CSP
Require: PA and T – as ordered lists of literals
1: **if** PA is **not** lex-least in its orbit under G **then**
2: result = false
3: **else**
4: $D \leftarrow t \in T$ if added to PA would make PA **not** lex-least
5: result = true and D
6: **end if**

The test proceeds by recursive search, similar to that described in [Linton, 2004], terminating when either the elements of PA have been exhausted, or the group at the bottom of the stabiliser chain consists only of the identity permutation. This stabiliser chain is the sequence stabiliser of the literals involving the decisions made above the current node during search. More precisely, a recursive routine with the following specification is applied:

Require: G – a permutation group
Require: $SOURCE$ and $EXTRA$ – as ordered lists of points
Require: $TARGET$ – an ordered list of points
1: **if** $\exists g \in G : g(SOURCE) \prec_{\text{lex}} TARGET$ **then**
2: result = false
3: **else**
4: $D \leftarrow \{t \in EXTRA : \exists g \in G : g(SOURCE \cup \{t\}) \prec_{\text{lex}} TARGET\}$
5: appropriate subset of D is added to a global list DL
6: result = true
7: **end if**

Calling this routine with the same G, and with $SOURCE$ and $TARGET$ both equal to PA and $EXTRA$ equal to T clearly achieves the specification above. The implementation of this routine is:

1: **GAPLexSearch**$(G, SOURCE, TARGET, EXTRA)$
2: **if** $TARGET$ is empty **then**
3: return true
4: **end if**
5: $x \leftarrow TARGET[1]$
6: **for** $y \in SOURCE$ **do**
7: **if** $\exists g \in G : g(y) < x$ **then**
8: **return** false
9: **else**
10: **if** $\exists g \in G : g(y) = x$ **then**
11: $G' \leftarrow$ the stabiliser of x in G
12: $S' \leftarrow SOURCE \setminus \{y\}$
13: $T' \leftarrow TARGET \setminus \{x\}$
14: $res \leftarrow \text{GAPLexSearch}(G', S', T', EXTRA)$
15: **if** res = false **then**
16: **return** false
17: **end if**
18: **end if**
19: **end if**
20: **end for**
21: **for** $y \in EXTRA$ **do**
22: **if** $y \notin DL$ and $\exists g \in G : g(y) < x$ **then**
23: add y to DL
24: **end if**
25: **end for**
26: **return** true

2.4 Empirical evaluation

Our implementation uses the GAP–ECL^{i}PS^e system, with CSP modelling and search performed in ECL^{i}PS^e, and with GAP providing black-box answers to symmetry questions. We have tested our implementation on two classes of CSP. The first is Balanced Incomplete Block Designs (BIBDs), problem class 28 in *csplib*. This class was chosen to be a stern test of the effectiveness of GAPLex, with a large number of symmetries and small domains. We would therefore expect the search for lex-inspired early backtracks to be expensive, with not many useful domain deletions being returned. This expectation is realised in our results given in Table 1. The better results for GAP-SBDD are in part because GAP-SBDD has special support for problems with Boolean variables. We tried posting the complete set of lexicographic lex-leader constraints on each BIBD instance, but the number of constraints was too great.

The second problem class is Graceful Graphs, a graph labelling problem described in [Petrie and Smith, 2003]. The symmetries that arise are any symmetries of the graph, combined with symmetries of the labels. In this class the domains are larger, and, in general, there are fewer symmetries.

Table 1: GAP-SBDD *vs* GAPLex. Problem class: BIBDs modelled as binary matrices.

					GAP-SBDD		GAP-LEX		Double Lex		GAP-Lex no prop		Combined	
V	B	R	K	λ	◇	◯	◇	◯	◇	◯	◇	◯	◇	◯
7	7	3	3	1	3	470	3	1150	3	20	21	1389	3	1150
6	10	5	3	2	4	869	29	80100	5	30	29	80100	4	50340
7	14	6	3	2	13	502625	-	-	30	110	-	-	-	-
9	12	4	3	1	12	451012	-	-	30	120	-	-	-	-
11	11	5	5	2	11	68910	-	-	20	140	-	-	-	-
8	14	7	4	3	14	219945	-	-	143	720	-	-	-	-

- $>$ 2 hours ◇ Number of Backtracks ◯ Total runtime in ms

Table 2: Table comparing various symmetry breaking methods. Partial GAP-LEX is where GAP-LEX checks do not commence until after the 1^{st} backtrack.

	GAP-SBDS				GAP-SBDD			
Instance	◇	□	△	◯	◇	□	△	◯
$K_3 \times P_2$	9	290	110	400	22	310	180	490
$K_4 \times P_2$	165	1140	3590	4730	496	3449	8670	12110
$K_5 \times P_2$	4390	35520	166149	201669	17977	174180	501580	675760
	GAP-LEX				Partial GAP-LEX			
Instance	◇	□	△	◯	◇	□	△	◯
$K_3 \times P_2$	10	160	100	260	12	150	130	280
$K_4 \times P_2$	184	1550	4020	5570	202	670	4980	5650
$K_5 \times P_2$	4722	47870	176200	224070	5024	18820	224310	243130

◇ Number of Backtracks □ Gap Time in ms △ Eclipse time in ms ◯ Total runtime in ms

The results for this class of problems (Table 2) are more encouraging. We see that, in contrast to BIBDs, GAPLex provides fewer backtracks but performs faster than GAP-SBDD. GAPLex performs as well as GAP-SBDS on these problems. We also tested the heuristic observation that no GAPLex tests will fail (resulting in a backtrack) until the first search-related backtrack occurs, although propagation may occur. The test involved simply turning GAPLex tests off until the first (if any) backtrack occurring in normal search. Our results for this heuristic are inconclusive for this class of problems.

3 Combining GAPLex with Incomplete Static SB methods

3.1 Double-lex

Much research has concentrated on symmetry-breaking constraints for *matrix models* – a constraint program that contains one or more matrices of decision variables – which occur frequently as CSPs. The prime example of this body of work is "double lex", which imposes that both the rows and the columns are lexicographically ordered [Flener *et al.*, 2002]. This does *not* break all the compositions of the row and column symmetries.

Frisch *et al* introduced an optimal algorithm to establish generalised arc-consistency for the $\preceq_{\text{lex}}$ constraint [Frisch *et al.*, 2002]. This gives an attractive point on the tradeoff: a linear time to establish a high level of consistency on constraints which often break a lot of the symmetry in matrix models. The algorithm can be used to establish consistency in any use of $\preceq_{\text{lex}}$, so in particular is useful for any use of lex-leader constraints.

3.2 Combining GAPLex and double-lex

Our approach is straightforward. We add static double-lex constraints before search. At each node in the search tree we run GAPLex, *w*ithout supplying the list of candidate domain deletions. This clearly means that the test is computationally more efficient: the final for-loop in the CGT algorithm isn't performed. We justify this with the hypothesis that any safe deletion found would almost certainly be already ruled out by the static double-lex constraints.

In Table 3 we see that GAPLex does not perform as well as simply posting double-lex constraints before search. However, GAPLex returns the correct number of solutions, whilst double-lex returns many symmetrically equivalent solutions. It seems that combining GAPLex with double-lex is a win over just using GAPLex. These results are not unexpected, as GAPLex was shown to behave poorly on this formulation of BIBDs in Section 2.4. We feel that the proof of concept is, however, interesting and useful.

3.3 Combining GAPLex with Puget's all-different constraints

Puget has recently presented a method of implementing lex-leader constraints for variable symmetries in CSPs with all-different constraints [Puget, 2005] in linear time. In problems with both variable and value symmetries, these can be usefully combined with GAPLex.

Table 3: Static symmetry breaking all-different constraints *vs* GAPLex with no search for safe deletions *vs* combined GAPLex and static constraints. Problem class: all solutions of Graceful Graphs in the standard model.

	Constraints		GAP-Lex no Prop				Combined			
Instance	◇	◯	◇	□	△	◯	◇	□	△	◯
$K_3 \times P_2$	16	800	12	150	130	280	10	140	100	240
$K_4 \times P_2$	369	4530	202	600	5140	5740	188	510	3300	3810
$K_5 \times P_2$	9887	297880	5024	19010	224740	243750	4787	14820	188820	203640

◇ Number of Backtracks □ Gap Time in ms
△ Eclipse time in ms ◯ Total runtime in ms

Our approach is the same as for double-lex: we post the static all-different constraints before search, and run the GAPLex test at each search node.

Our results, given in Table 3, show that GAPLex performs slightly better than static constraints, and that combining the two methods is better than using either in isolation. These encouraging results are made better by noting that, for this class of problems, using only static constraints results in twice as many solutions being returned as necessary.

4 Conclusions and Future Work

We have used and extended recent advances in Computational Group Theory to add lex-ordering to the class of symmetry breaking techniques that can be effectively implemented by using a CGT system to provide black-box answers to symmetry related questions. Our implementation, GAPLex, is competitive with GAP-SBDS and GAP-SBDD. The choice of which method to use for which class of CSPs appears to be an interesting research question. Answers to this question could provide insight into yet more symmetry breaking methods. Also the CGT algorithms used are still new and based on experience with GAP-SBDD may improve by orders of magnitude with further investigation.

We have, moreover, demonstrated the first combination of static and search-based symmetry breaking methods that is (for certain classes of CSP) more efficient than using either the static or search-based method in isolation. This result is important, since the successful combination of symmetry breaking methods is taxing and open area of CSP research, with great potential benefits attached to positive answers.

More work is needed in two areas. Firstly, we must address the questions relating to why a particular symmetry breaking approach works better for some CSPs than others. Secondly, we need to investigate other potentially successful combinations of symmetry breaking techniques.

Acknowledgements

We are very grateful for their helpful comments and other assistance to Ian P. Gent and Barbara Smith, and all the attendees at the 2006 Symnet sandpit on Combining Symmetry Breaking Techniques. This work is supported by EPSRC grants GR/S30580/01 (Symmetry and Inference), EP/CS23229/1 (Critical Mass) and GR/S86037/01 (Symnet Network).

References

[Backofen and Will, 1999] R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Proceedings, CP 99*, pages 73–87. Springer, 1999.

[Bosma and Cannon, 1993] W. Bosma and J. Cannon. *Handbook of MAGMA functions*. Sydney University, 1993.

[Brown *et al.*, 1988] C.A. Brown, L. Finkelstein, and P.W. Purdom, Jr. Backtrack searching in the presence of symmetry. In T. Mora, editor, *Proc. AAECC-6*, pages 99–110. Springer-Verlag, 1988.

[Crawford *et al.*, 1996] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. KR'96*, pages 149–159, November 1996.

[Fahle *et al.*, 2001] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Proc. CP 01*, pages 93–107, 2001.

[Flener *et al.*, 2002] P. Flener, A. M. Frisch, B. Hnich, Z. Kızıltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Proc. CP 02*, pages 462–476. Springer-Verlag, 2002.

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

[Frisch *et al.*, 2002] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Global constraints for lexicographic orderings. In *Proc. CP 02*, pages 93–108. Springer, 2002.

[Frisch *et al.*, 2004] A. M. Frisch, C. Jefferson, and I. Miguel. Symmetry breaking as a prelude to implied constraints: A constraint modelling pattern. In *Proc. ECAI 04*, pages 171–175, 2004.

[GAP, 2000] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.2*, 2000. (`http://www.gap-system.org`).

[Gent and Smith, 2000] I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In *Proc. ECAI 2000*, pages 599–603, 2000.

[Gent *et al.*, 2002] I. P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In *Proc. CP 02*, pages 415–430, 2002.

[Gent *et al.*, 2003] I. P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD using computational group theory. In *Proc. CP 03*, pages 333–347, 2003.

[Kelsey *et al.*, 2004] T. Kelsey, S. Linton, and C. M. Roney-Dougal. New developments in symmetry breaking in search using computational group theory. In *Proc. AISC 04*, pages 199–210, 2004.

[Linton, 2004] S. Linton. Finding the smallest image of a set. In *Proc. ISSAC 04*, pages 229–234. ACM Press, 2004.

[Meseguer and Torras, 2001] P. Meseguer and C. Torras. Exploiting Symmetries within Constraint Satisfcation Search. *AI*, 129:133–163, 2001.

[Petrie and Smith, 2003] K. Petrie and B. Smith. Symmetry breaking in graceful graphs. In *Proc. CP 03*, 2003.

[Puget, 1993] J-F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Proc. ISMIS 93*, pages 350–361, 1993.

[Puget, 2003] J-F. Puget. Symmetry breaking using stabilizers. In *Proc. CP 2003*, pages 585–599, 2003.

[Puget, 2005] J-F. Puget. Breaking symmetries in all different problems. In *Proc. IJCAI 05*, pages 272–277, 2005.

[Roney-Dougal *et al.*, 2004] C. M. Roney-Dougal, I. P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In *Proc. ECAI 04*, pages 211–215, 2004.

[Wallace *et al.*, 1997] M. G. Wallace, S. Novello, and J. Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12(1):159–200, May 1997.

Solving Partially Symmetrical CSPs

F. Verroust and N. Prcovic - LSIS - Université Paul Cézanne - Aix-Marseille III
fverroust@lsis.org, nicolas.prcovic@lsis.org

Abstract

Many CSPs contain a combination of symmetrical and asymmetrical constraints. We present a global approach that allows to apply any usual methods for breaking symmetries on the symmetrical part of a CSP and then to search for a global solution by integrating afterwards the asymmetrical constraints. Then, we focus on optimization problems where only the cost function is asymmetrical. We show experimentally that in this case we can speed up the search of some problems.

1 Introduction

Symmetry breaking for CSPs has been largely studied for the last ten years. When a CSP contains many constraints, avoiding to explore the large portions of the search space containing equivalent partial assignments greatly allows to save time.

The CSP formalism allows a constraint to be defined as any relation between many variables, more precisely as any subset of the cartesian product of their domains. Therefore, very few constraints are likely a priori to exhibit symmetries. However, practical problems containing symmetries are far from being insignificant, despite minority all the same. Actually it turns out that many CSPs contain a combination of symmetrical constraints (e.g., equality, difference, sum of variables equal to a constant) and asymmetrical constraints. These CSPs are then globally asymmetrical and cannot directly take profit from the usual symmetry breaking methods. The main point of this paper is to propose a general scheme for handling these partially symmetrical CSPs and thus extend the application of the current symmetry breaking methods. This idea to handle symmetrical constraints separately recently appeared in [Martin, 2005] and [Harvey, 2005].

In section 2 we will show the interest of this approach through the example of a simple CSP. In section 3, we will remind several notions about symmetry groups, which will allow us to formally present our global resolution scheme in section 4. Then, we will study in section 5 the more specific case of a certain type of optimization problems, where our method is likely to be specially efficient. We will experiment it out on two kind of problems in section 6.

2 An example

We first give a general idea of our resolution scheme through one very simple example. Consider a CSP P with 3 variables x, y and z, each defined on the same domain $\{1, 2, 3\}$. The problem P contains only two constraints: $xyz = 6$ and $x + 2y + 3z = 10$. This problem has a single solution: $x = 3$, $y = 2$ and $z = 1$. It has no symmetry. The size of the search space is composed of $3^3 = 27$ combinations of values.

Now consider the problem P', which is the same problem as P but only keeping the first constraint $xyz = 6$. P' contains all the possible variable symmetries: from any solution, another can be obtained by permuting variables. Precisely, the set of the six solutions of P' is $x = 1$, $y = 2$ and $z = 3$ and any variable permutation (swapping x and y, swapping x and z or any composition of the two swappings). A symmetry breaking method can find a solution quickly. For instance, techniques adding symmetry breaking constraints before the search (e.g., [Puget, 2005b]) allows to post the constraints $x < y$ and $y < z$ that break all the symmetries. The only remaining canonical solution is then $x = 1$, $y = 2$ and $z = 3$[1].

We know that the set of solutions of P is included into the one of P'. To obtain a solution of P, it suffices to enumerate the solutions of P' and to check whether they respect the constraint $x + 2y + 3z = 10$ or not. Enumerating the solutions of P' is trying all the variable permutations of its canonical solution. The expected saved time is based on the fact that the search space of P is not anymore the set of all the combinations of the variable assignments (size: $3^3 = 27$) but the set of the (canonical and non canonical) solutions of P' (size: $3! = 6$), which is much smaller. Our resolution scheme can only be efficient if the time spent computing the canonical solutions of P' and enumerating the symmetrical solutions of P' is shorter than solving P in a usual way.

Before presenting our resolution scheme in a general frame, we will recall some useful notions about symmetries and computational group theory.

[1] Notice that thanks to the constraints breaking symmetries, a simple application of arc consistency reduces the domains to one value: $x < y$ allows to eliminate 1 from the domain of y and 3 from the domain of x, then $y < z$ allows to eliminate 1 and 2 from the domain of z, and so on.

3 Preliminaries

We remind here usual definitions and notations on permutation groups, which can be found in [Seress, 1999] for instance. In a mathematical sense, a *group* is a set structured by a binary associative, inversible operator $\circ$ such that G is closed ($\circ$ that maps any pair of elements of G to an element of G) and contains the neutral element e for $\circ$. H is a subgroup of G, noted H $\leq$ G iff H is a subset of G and H is a group for $\circ$.

A *permutation* is a one-to-one mapping of a set to itself. A permutation is described by a set of *cycles* of the form (ω_1 ω_2 ... ω_k), which means $\forall \omega_j$, ω_j maps to ω_{j+1} and ω_k maps to ω_1 (e.g., 2, 4, 5, 1 and 3 are the images of 1, 2, 3, 4 and 5 by the permutation (1 2 4)(3 5)). A permutation can also be applied to a set or a tuple (e.g., for the permutation (1 2 4)(3 5), the image of the pair (1,5) is (2,3), the image of (1,2,3,4,5) is (2,4,5,1,3), and the image of the set $\{2, 3, 4\}$ is $\{1, 4, 5\}$).

Roughly, a permutation of the elements of Ω that preserves the relations involving these elements (ie, relations that are still true for their images) is a symmetry (or *automorphism*). The set of symmetries of Ω is a group G for the binary operator $\circ$ (of composition). We say that G *acts* on Ω. To avoid ambiguities, we call *points* the members of Ω and we keep the word *element* for the members of G. If $\sigma \in$ G and $\omega \in \Omega$, we denote ω^σ the image of the point ω by the symmetry σ. $\circ$ is the only operator applyable on symmetry, so we will write $\sigma_1\sigma_2$ instead of $\sigma_1 \circ \sigma_2$.

Definition 1 *CSP microstructure*
A CSP microstructure (Ω, A) is a (hyper)graph where each vertex corresponds to a variable assignment and each (hyper)edge corresponds to a possible tuple of values for a constraint.

Definition 2 *Symmetry of a CSP*
Let (Ω, A) be a CSP microstructure. A symmetry of a CSP is a permutation of Ω which, applied to A, leaves A unchanged.

We deal with the most general possible symmetries, not only restricted to variable symmetries (the values of an assignment are preserved, not the variables) or value symmetries (the variables are preserved, not the values): applied on a (partial or complete) assignment, all the variables and values can change. This corresponds to the *syntactical symmetry* definition in [Benhamou, 1994] or the *constraint symmetry* definition in [Cohen *et al.*, 2005].

Figure 1 presents a 4×4 grid of 16 points. Each point is a variable assignment. But we do not need to know them to apply the method we are presenting. It can represent the chessboard of the four-queen problem (the points 1, 2, 3, 4, 5, ... are the variable assignments $x_1 = 1$, $x_1 = 2$, $x_1 = 3$, $x_1 = 4$, $x_2 = 1$, ...) or any CSP where the sum of the domain sizes is equal to 16 (e.g., a CSP with two variables of domain size 4 or a CSP with 8 boolean variables) and which has the same symmetries.

Definition 3 *Orbit*
The orbit ω^G of the point ω in Ω on which the group G acts is $\omega^G = \{\omega^\sigma : \sigma \in G\}$, i.e. all possible images of ω by a permutation of G. This notion can be extended to a set of points $\Delta \subseteq \Omega$: $\Delta^G = \{\{\omega^\sigma : \omega \in \Delta\} : \sigma \in G\}$.

In the example in figure 1, $1^G = \{1, 4, 13, 16\}$, $2^G = \{2, 3, 5, 8, 9, 12, 14, 15\}$, $\{1, 2\}^G = \{\{1, 2\}, \{1, 5\}, \{3, 4\}, \{4, 8\}, \{9, 13\}, \{13, 14\}, \{12, 16\}, \{15, 16\}\}$.

In this paper, we deal with orbits of microstructure vertices (variable assignment) or orbits of set of vertices (partial assignments, which implies several variables). If I=$\{\omega_1, \omega_2, ..., \omega_i\}$ is a partial or complete assignment then the set of symmetrical assignment I^G is $\{I^\sigma : \sigma \in G\}$, i.e. $\{ \{\omega_1^\sigma, \omega_2^\sigma, ..., \omega_i^\sigma\} : \sigma \in G\}$. When I is a CSP solution, I^G is a set of CSP (symmetrical) solutions, whereas if I is not a solution, then I^G contains no solution.

Definition 4 *Generators of a group*
A generating set *of the group G is a subset H of G such that each element of G can be written as a composition of elements of H. We write G=<H>. An element of H is called a* generator.

The eight symmetries of a grid (see figure 1) can be generated by two generators: the vertical reflection γ_1 and the diagonal reflection γ_2. All the symmetries can be expressed as compositions of γ_1 and γ_2: e , γ_1, γ_2, $\gamma_2\gamma_1$, $\gamma_1\gamma_2$, $\gamma_2\gamma_1\gamma_2$, $\gamma_1\gamma_2\gamma_1$ and $\gamma_1\gamma_2\gamma_1\gamma_2$.

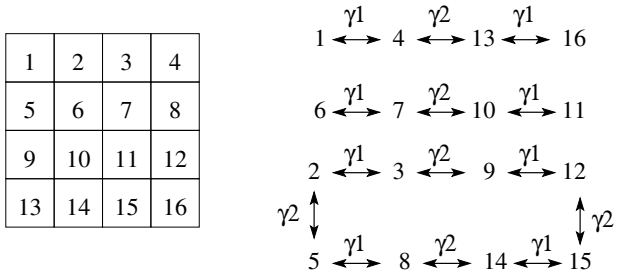


Figure 1: A 4×4 grid and its orbit graph. $\Omega = \{1, 2, ..., 16\}$. G=<$\{\gamma_1, \gamma_2\}$> with $\gamma_1 = \{(1\ 4)(2\ 3)(5\ 8)(6\ 7)(9\ 12)(10\ 11)(13\ 16)(14\ 15)\}$ and $\gamma_2 = \{(2\ 5)(3\ 9)(4\ 13)(7\ 10)(8\ 14)(12\ 15)\}$.

An important property of a *strong* generating set is that its cardinality is bounded by a pseudolinear function of $|\Omega|$, whereas the order of G is bounded by $|\Omega|!$. If we know a CSP solution and each element of the permutation group G, we can compute all the symmetrical solutions by applying once each element of G. G can be too large for a computer memory whereas a strong generating set has a moderate size and allows to compute all the symmetrical solutions, applying all possible compositions of generators to the solution.

Definition 5 *Pointwise stabilizer*
A pointwise stabilizer $G_{(\Delta)}$ *of $\Delta \subseteq \Omega$ is the subgroup $G_{(\Delta)} = \{\sigma \in G : \forall \omega \in \Delta, \omega^\sigma = \omega\}$, i.e. the set of symmetries of G that fix each point of Δ.*

In the example of figure 1, G = <$\{\gamma_1, \gamma_2\}$>, $G_{(1)}$ = <$\{\gamma_2\}$>, $G_{(1,2)}$ = $\{e\}$.

A permutation group can be represented intentionnally by a base B=($\beta_1, ..., \beta_k$), which is a sequence of points of Ω. B is a base for G iff the only pointwise stabilizer of B in G is the identity, i.e: $G_{(B)}$ = {e}. B defines a chain of pointwise

stabilizers:

$$G=G^{[1]} \geq G^{[2]} \geq ... \geq G^{[k]} \geq G^{[k+1]} = \{e\}$$

where $G^{[j]} = G_{(\beta_1,...,\beta_{j-1})}$. This base allows to find elements of G to generate successively $G^{[1]}$,...,$G^{[k+1]}$. The Schreier-Sims algorithm [Seress, 1999] constructs a set of generators $\{\gamma_i^{(j)} : 1 \leq j \leq k, 1 \leq i \leq t_j\}$ of G, called a *strong generating set*, such that:

$G_{(\beta_1,...,\beta_{h-1})} = <\{\gamma_i^{(j)} : h \leq j \leq k, 1 \leq i \leq t_j\}>$

In other words, the generators $\gamma_1^{(j)}, ..., \gamma_{t_j}^{(j)}$ fix the points β_1,... β_{h-1} but not β_h. A tool such as Nauty [McKay, 1981] computes a base and a strong generating set from a vertex-colored graph. Thus we can obtain automatically a compact representation of a symmetry group from a CSP microstructure.

Definition 6 *Orbit graph*
An orbit graph *is a directed graph were each vertex is a point of* Ω*. For any vertices* i *and* j*, an arc* (i,j) *exists, and is labeled with* γ*, iff* $j = i^\gamma$ *and* γ *is a generator of G.*

Thanks to an orbit graph, some questions on permutation groups are reducible to questions on graphs. For example, vertices in the same connected component of an orbit graph are in the same orbit (cf figure 1).

4 The general resolution scheme

Consider a CSP, called P, containing n variables, with finite domains and contraints of any arity. We make a partition of the constraint set C in two sets C_{sym} and C_{rest}. Let us call P_{sym} the CSP that corresponds to the problem P where the constraints of C_{rest} are removed so as to keep only the ones of C_{sym}.

Our resolution scheme is interesting if C_{sym} is chosen such that P_{sym} contains symmetries. It is not always easy to acheive it. However, in practice, most constraints have their own semantics. So, it is easy to know which locally induce symmetries. Furthermore, [Puget, 2005a] presents general methods for agregating these symmetric constraints so as to have a graph representation for them, from which global symmetries can be derived. So there is actually many practical cases where this partition can be easy to perform, or even automated.

Searching for a solution of P is performed, on the one hand, searching for canonical solutions of P_{sym}, and on the other hand, exploring the orbit of each canonical solution of P_{sym} in order to find one that also respects the constraints of C_{rest}.

To solve P_{sym}, any existing symmetry breaking techniques may be used. We just have to modify slightly the algorithm: as soon as a canonical solution I of P_{sym} is found, we explore the orbit of I so as to find a symmetrical solution that respects the constraints of C_{rest}. If one is found, we stop because it is a solution of P. If not, we let the search for other solutions of P_{sym} continue. Notice that this resolution scheme was recently proposed in [Harvey, 2005]. However, no concrete, efficient way of exploring the orbit was described in this paper. The method described in [Martin, 2005] is roughly equivalent. It uses additional variables that are constrained in order to be assigned to a symmetrical solution of P_{sym}. So, these variables represent a solution of the orbit of one solution found for P_{sym}. No clue is given on how to build the constraints linking the P_{sym} variables and the additional variables. How to explore efficiently an orbit is thus left aside in these two papers and this is the key problem we want to tackle. We can consider a systematic or incomplete exploration of the orbit of I.

4.1 Local search in an orbit

When the orbit of a canonical solution of P_{sym} is large, we can consider exploring only a part of it, using a local repair method. Though, we can easily fit any metaheuristic method (Min-conflicts, tabu search, simulated annealing,...), considering the neighbor of a complete assignment does not result from changing the value of a single variable, but from the application of a generator. Thus, the neighborhood of a complete assignment I is $\{ I^\gamma : \gamma \in H\}$ if G=<H>. The complete assignment is evaluated counting the number of conflicts in C_{rest}.

We can guide the search with a heuristic for selecting the most promising generators. A first heuristic is to simply choose the generator that decreased the greatest number of conflicts. Another heuristic is to select the generator that fixes the greatest number of points (and lowers the number of conflicts). Such a heuristic improves a complete assignment by modifying as few values as possible, like in a usual local search. The first heuristic is likely to decrease quickly the number of conflicts but to reach soon a local minimum. The second heuristic is more cautious, trying smaller improvements but a longer time.

4.2 Systematic search of an orbit

Now if we wish to enumerate all the symmetrical solutions of the orbit of a canonical solution to P_{sym}, we have to be able to find all the symmetries of the group G of the microstructure of P_{sym} from its set of generators.

One possible method to enumerate all the permutations of G is to make a tree search where each node of the tree holds a permutation. We obtain all the children of a node applying each generator to the permutation. We memorize all the permutations as they are produced, checking each of them to know if we had already found it. In this case, we leave the branch (we backtrack if it is a depth-first search). The main drawback of this method is the space complexity, which is of the order (the number of elements) of the group, at worst equal to $|\Omega|!$. This method requires to memorize all the permutations for the following reason. We can represent each permutation by a word which symbols are the names of the generators (e.g., the U-turn of the grid of figure 1 can be represented by the word $\gamma_1\gamma_2\gamma_1\gamma_2$). It is easy and low-memory consuming to enumerate words. However, several words can represent the same permutation (e.g., $\gamma_2\gamma_1\gamma_2\gamma_1$ also represents the U-turn). So, we if want to avoid redundancy, we have to compute and memorize permutations instead of words.

Actually, there exists a much efficient and classical algorithm for computing orbits, presented for instance in [Seress, 1999]. However, our point is not to generate all the permutations of a canonical solution to P_{sym}, but only a permutation

whose image is a solution to P (ie, respecting the constraints of C_{rest}). A generate and test algorithm would be really inefficient. Now, we present an adaptation of the classical algorithm for computing as few permutations as possible.

Permutation tree

Consider a permutation group G, a base $(\beta_1, ..., \beta_k)$ of G and his strong generators $\Gamma=\{\gamma_i^{(j)} : 1 \leq j \leq k, 1 \leq i \leq t_j\}$, where the group induced by $\{\gamma_1^i, ..., \gamma_{t_i}^i, ..., \gamma_1^k, ..., \gamma_{t_k}^k\}$ is $G^{[i]} = G_{(\beta_1,...,\beta_{i-1})}$, the stabilizer of $(\beta_1, ..., \beta_{i-1})$. Any permutation is going to move some points, and leave others fixed. Due to the stacking of the stabilizers, determined both by the base and the partition of his strong generators, we can still cut a permutation into several permutations moving only a part of the points. More precisely, any permutation can be expressed as $\sigma_1\sigma_2...\sigma_k$, where each σ_i is a permutation (composed of several generators) moving β_i and possibly other points but leaving fixed the points $\beta_j, \forall j < i$. The permutations σ_i are those of the stabilizer $G_{(\beta_1,...\beta_{i-1})}$ (and are moving β_i). Any permutation σ_i can be expressed as a composition of generators of the set $\{\gamma_h^{(j)} : i \leq j \leq k, 1 \leq h \leq t_j\}$. Enumerating the orbit of a canonical solution applying to it all the permutations of G will then consist of a depth-first search. We start from the canonical solution applying it all the permutations of the form σ_1 (including the identity), then to each of them we apply all the permutations of the form σ_2, and so on, up to σ_k. The set of leaves of the search tree represents the orbit of the canonical solution. We now have to explain how to determine all the permutations of the form σ_i.

Consider the orbit graph of G. Any path of this graph starting with β_i is labeled by a sequence of generators forming a word which, on the one hand, corresponds to a permutation moving β_i, and on the other hand, starts with a generator belonging to $\{\gamma_h^{(i)} : 1 \leq h \leq t_i\}$. These paths can be found thanks to a depth-first search in the orbit graph starting at the point β_i. The set of paths thus corresponds to the words representing the permutations which are moving β_i. If now we remove from the orbit graph all the arcs whose label stands on an arc whose end is part of the set $\{\beta_1, \beta_2, ... \beta_{i-1}\}$, the set of paths left, starting with β_i, are corresponding to the permutations leaving fixed the points of $\{\beta_1, \beta_2, ... \beta_{i-1}\}$. They represent the words of the form σ_i we were looking for.

Figure 2 shows the search tree of the orbit of the set of points $\{1, 2, 14\}$.

Choice of the base

The interest to have a strong generating set built from a base is we do not fix all the points each time we step off a level in the search tree. Now, a point corresponds to a variable assignment. Thus it is sure this variable is not going to have its value changed in the subtree where its point was fixed. Notice that at each level of the search tree, the fact of fixing a point often leads to fixing other points. On the orbit graph of $G_{(1)}$ on figure 2, we can see that fixing point 1 has also fixed points 6, 11 and 16. Knowing the set of the variables whom we know are not going to have their values changed allows us to check the constraints of C_{rest} (or to apply mechanisms of domain filtering) and to backtrack in case of inconsistency without having to explore the subtree. Backtracking at depth i allows not

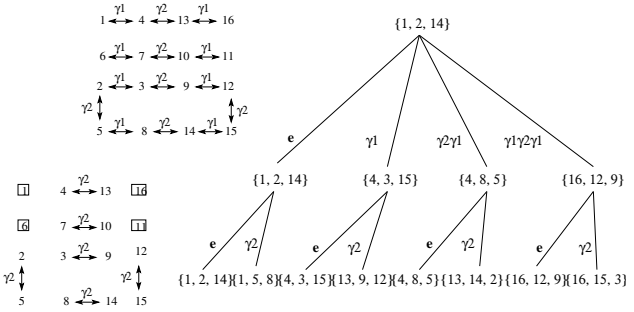


Figure 2: On the right, the search tree producing the orbit of $\{1, 2, 14\}$. On the left, two orbit graphs. The one on the top is the orbit graph of G and allows to produce the 4 beginnings of permutation of the form σ_1 of the first level of the tree. The one on the bottom is the orbit graph $G_{(1)}$ and allows to produce the ends of permutations of the form σ_2 of the second level. The leaf $\{4, 8, 5\}$ allows to determine that applying the permutation $\gamma_2\gamma_1 e = \gamma_2\gamma_1$ we can find again the root of the tree $\{1, 2, 14\}$.

to search the subtree containing all the permutations of the form $\sigma_{i+1}...\sigma_k$ that could complete the permutation $\sigma_1...\sigma_i$ we reached. The size of the base is equal to the maximum depth of the tree, which is the number of steps where some constraints can be checked before a permutation of the form $\sigma_1\sigma_2...$ be complete. Therefore, we have to choose the base containing the more important number of points, as we have more often the possibility to eliminate subtrees corresponding to completions of permutations.

Filtering

In addition, to examine an orbit graph allows to know the set of values each variable can be assigned to, which means its domain of possible values. The union of the orbits of the points of a solution represents the points which can appear in the symmetrical solutions. Some points are part of no orbit and can be thus removed of the domains of the variables. For instance, if the set of points $\{1, 2, 14\}$ (see figure 2) is a solution of a CSP, then the points 6, 7, 10 and 11 are not reachable by a permutation and can be removed from the domains. This domain filtering can be completed each time we are moving in the depth of the search tree. Actually, the orbit graph looses arcs (as points are fixed) and new points are becoming unreachable. For instance, in the node of depth 1 of the search tree in figure 2 containing the set of points $\{1, 2, 14\}$, we can again eliminate the points 4, 13, 3, 9, 12, 15 and 16 which have become unreachable through the points 1, 2 or 14. Of course, removing this way a value from a domain can allow to remove other values from the constraints of C_{rest}, applying usual methods of propagation (e.g., arc consistency). Complementarily, removing a value by filtering the constraints of C_{rest} can eliminate some points of the orbit graph and thus fix other points. For instance, if we consider the node of the last example, eliminating point 5 from the orbit graph (of the second level) by constraint propagation fixes point 2. No permutation can thus be applied to $\{1, 2, 14\}$: it becomes useless to search the subtree from this node.

The interaction between these two types of domain filtering is complex to grasp. Adapting the techniques for maintaining forms of local consistency is a vast topic that will need to be studied to make more efficient the search of the orbits of the canonical solutions of P_{sym}.

4.3 Complexity of the search in an orbit

The time expectedly gained is based on the fact the size of the search space of P is higher or equal to the sum of the sizes of all the orbits of the canonical solutions of P_{sym}. Actually, the orbits contain the set of solutions (whether they are canonical or not) of P_{sym}, which is potentially smaller than the search space of P (which is the set of combinations of values of the problem).

It is not possible to evaluate in a general case the size ratio between the search space of P and the orbit size of a canonical solution of P_{sym}. But it can be done in the case there are only variable symmetries. In this case, the symmetrical solutions of the orbit will contain the same values but differently distributed among the variables.

A CSP P with n variables of domains of size d has a search space of d^n combinations of values. Let us calculate now the maximum size of the orbit of a canonical solution of P_{sym}. In the worst case about the size of the orbit, any permutation of variables is a symmetry of the group. There are m different values in the solution, with $m \leq d$ and $m \leq n$. Call v_i the number of occurrences of the i^{th} value in a solution. The size of the orbit is T(n, m) = $\frac{n!}{\prod_{1 \leq i \leq m} v_i!}$ ($n!$ is the number of permutations of n elements we have to divide by each $v_i!$, the number of permutations uselessly swapping the same values). As we have the relation $\sum_{1 \leq i \leq m} v_i = n$, we minimize the product $\prod_{1 \leq i \leq n} v_i!$ when the v_i have values as close as possible. In the worst case, all the values v_i equal to $\frac{n}{m}$. Thus T(n, m) $\leq \frac{n!}{((\frac{n}{m})!)^m}$. Approximating the factorials from the Stirling formula: $n! = \sqrt{2\pi} n^{n+\frac{1}{2}} e^{-n}(1 + \epsilon(n))$ where $\epsilon(n)$ tends to 0 when n is large, we get T(n, m) $\leq (\sqrt{2\pi})^{1-m} \frac{\sqrt{n}}{(\sqrt{\frac{n}{m}})^m} m^n \frac{1}{(1+\epsilon(n)^{m-1})}$. So we have T($n$, m) $\in O(\frac{m^{n+\frac{m}{2}}}{(2\pi)^{\frac{m}{2}} n^{\frac{m-1}{2}}})$. As it is not obvious to compare this complexity to d^n, we show in table 1 a comparison of the complexities according to a few values of d, taking the less favorable case $m = d$.

To have a global comparison between a classical resolution and our approach, we have to consider the resolution time of P_{sym}, which may still remains in $\Theta(d^n)$, and the fact P_{sym} can have a great number of canonical solutions and thus of orbits to explore. Our approach can be efficient a priori only if P_{sym} is quickly solvable and contains few canonical solutions.

Comparing the complexity of the search spaces is not accurate enough. Another important condition of efficiency is that C_{rest} does not help much to filter when P is solved in a standard way. When solving P_{sym}, we have removed C_{rest} and added constraints to obtain canonical solutions only. Solving P_{sym} can still be longer than solving P because filtering thanks to the constraints of C_{rest} also prunes the search tree.

d	d^n	T($n, m = d$)
2	2^n	$O(\frac{2^n}{\sqrt{n}})$
3	3^n	$O(\frac{3^n}{n})$
n	n^n	$O(\frac{\sqrt{n}}{(2\pi)^{\frac{n}{2}}} n^n)$

Table 1: Comparison between the size of the search space of a CSP and the orbit size of a canonical solution in the case of variable symmetries. For instance, the line $d = 3$ shows that if the complexity of computing the canonical solutions of P_{sym} is lower than $O(\frac{3^n}{n})$ and the number of these canonical solutions is lower than n, then the overall complexity of solving P is reduced.

5 A specific case of optimization

We show now a specific case of application of our method where it can possibly be efficient. An optimization problem can be described by a CSP with a cost function on the CSP variables. The point is to find the solution of the problem which minimizes the cost function. This type of problem can be solved using the usual Branch & Bound method (B&B). This method can be seen as performing the search of one solution and posting a constraint forbiding the cost function to exceed the cost of the solution. Then, this constraint is reactualized every time a new solution is found. If we deal with a symmetrical CSP whose cost function is asymmetrical, our method applies directly and simply: C_{sym} contains all the constraints and C_{rest} contains the constraint that the cost function must not exceed the cost of the current best solution. In this case, the best solution is searched in the orbits of the canonical solutions of P_{sym}. Our method is likely to be efficient because the cost functions usually involve many variables and does not help much to prune the search tree.

5.1 A complete version

In a complete version of our method, we must search each orbit with the depth-first search we described in the section 4.2. During the exploration of the orbit, filtering can be performed bounding the cost of the best symmetrical solution of the orbit if the cost function has good properties, for instance, monotony or linearity. The orbit graph shows the lowest and greatest values each variable can be assigned to. Agregating the variable bounds, we can check if the orbit cannot contain any better solution than the canonical solution. This can also be performed dynamically during the tree search. At each node, the fixed part of the solution being permuted gives an exact value of the corresponding part of the cost function. A lower bound can also be given to the non fixed part. Notice that the variable bounds are refined as we get deeper into the search tree because points become unreachable as some others are fixed, as we saw it in section 4.2.

5.2 A recompleted local version

If the order of the symmetry group is very large, we can consider performing a local search in the orbits, as proposed in section 4.1. A simple greedy algorithm where we apply iteratively the most promising generator (selected by any of the

two heuristics we proposed) may already find a suboptimal, but good enough, solution. However, there is an easy way to make this method complete. It suffices to let the B&B algorithm find all the solutions as usual (and not only the canonical ones thanks to symmetry breaking techniques). In this case, we explore partially the orbits in order to find a symmetrical solution that has a lower bound. We use this lower bound to help pruning during the remaining tree search. In other words, we have a standard B&B algorithm that always tries to find an even better solution than the current best solution it has just found, looking in its orbit before continuing the B&B search.

6 Experiments

The first problem we experimented was the *weighted magic square problem*, mentioned for instance in [Martin, 2005]. The goal is to fill a $n \times n$ grid with all the integers from 1 to n^2 such that the sum of each row, column and the two diagonals equal the same number. In addition, each field of the square has a weight and we have to minimize the sum of the values of the fields multiplied by their own weight (which makes the cost function linear). The weight of each field is chosen at random between 1 and $100n^2$. The order of the permutation group of the grid is 8 (the same as the one of a n-queen problem). Our program is written using Ilog Solver. From the CSP microstructure, we extract a base and a strong generating set of its permutation group thanks to Nauty [McKay, 1981].

We compared three techniques, the usual B&B, the usual B&B plus a greedy search in the orbits (called *GreedySym*) and B&B with a complete tree search in the orbits (called *TreeSearchSym*). For each size of problem, we ran 20 instances with different random values for the cost function and reported the average results (see figure 3 and 4). We can see that TreeSearchSym has converged very quickly to a near optimal solution for $n = 5$ and has found a better solution whithin 3 minutes for $n = 6$.

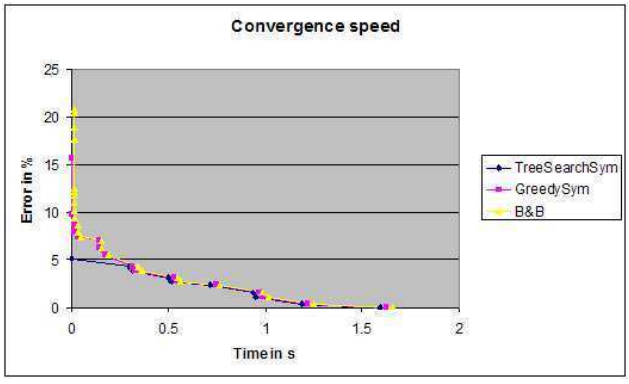


Figure 3: Convergence speed to the best solution for the weighted square problem of size 5. TreeSearchSym converges more quickly at the beginning but the curves coincide at last and the resolution time are equal.

The second problem of our experiments is a graph coloring problem. It contains the variables $\{x_1, ..., x_n\}$, n being a multiple of 5, for which values are in {0,1,2,3,4}. The constraint set is defined by:

- $\{x_i, x_{i+1}, x_{i+2}, x_{i+3}\}$ have different values.
- $\{x_i, x_{i+4}, x_{i+5}, x_{i+9}\}$ have different values (except if $i = n - 5$).

See figure 5 for an illustration of this problem.

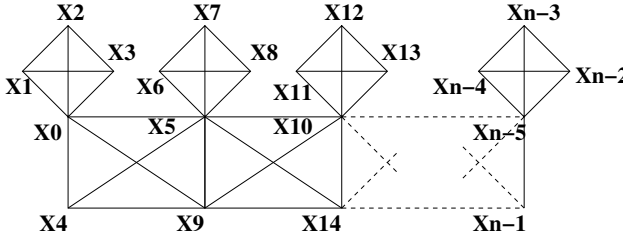


Figure 5: The graph to color

We chose this problem because, even if it is artificial, it has the advantage that its number of variables can easily be increased while keeping the same type of symmetries. Indeed, for any n multiple of 5, there exists two types of variable symmetry. The first one is local: for all i multiple of 5, the variables $\{x_{i+1}, x_{i+2}, x_{i+3}\}$ are interchangeable. The second type of symmetry is global. Consider the partition of the set of variables into k parts of 5 variables defined by: $\forall j \in [1; k]\{x_j, x_{j+1}, x_{j+2}, x_{j+3}, x_{j+4}\}$. These sets of variables are symmetrical by the reflection that exchange x_j and x_{k-j}, $\forall j$.

With this problem, the order of the permutation group grows exponentially with n. The cost function is a linear function of the n variables. Each coefficient associated to a variable is an integer chosen randomly between 1 and n.

The results are shown in table 2 and figure 6.

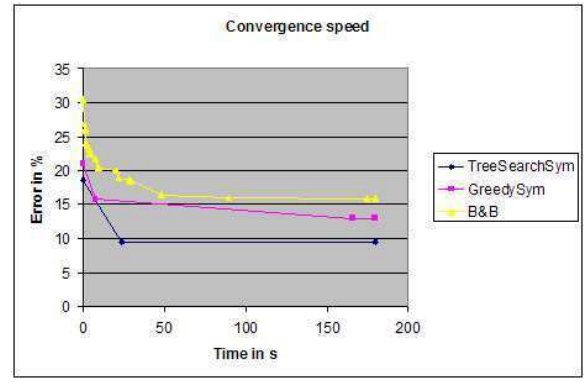


Figure 4: Convergence speed to the best solution for the weighted square problem of size 6. TreeSearchSym reaches faster a better solution than GreedySearch and B&B after 3 minutes. None completed their search after several hours. (At last, CPLEX, a mathematical programming optimizer of Ilog, was used to compute the value of the best solution and allowed us to know how far from the best solutions were the solutions we found.)

# of variables	TreeSearchSym	GreedySym	B&B
15	0.054	0.0035	0.0042
20	0.25	0.025	0.026
25	2.35	0.16	0.17
30	11.6	1.04	1.11
35	94.6	6.39	6.7
40	492	44.5	47.3

Table 2: Resolution times of the graph coloring problem.

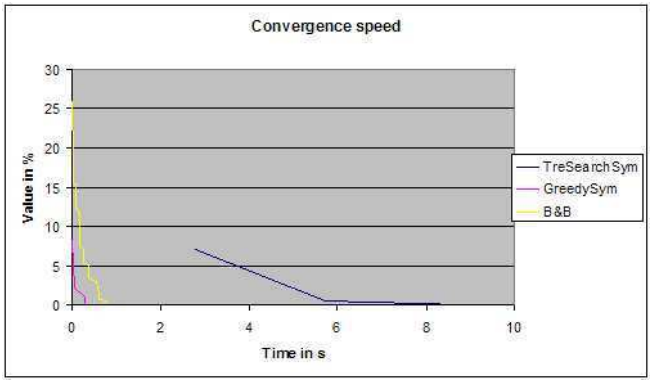


Figure 6: Convergence speed to the best solution for the graph coloring problem with $n = 30$. GreedySym converges faster than B&B. TreeSearchSym converges very slowly. The same behavior has been observed for the other values of n.

The resolution time of GreedySym is always a little faster ($<$10%) than the one of B&B but GreedySym reaches an optimal solution always much faster than B&B. TreeSearchSym has very bad performances because it spends a long time searching systematically the large orbits.

7 Conclusion and perspectives

Since CSPs often mix symmetrical and asymmetrical constraints in practice, giving methods for handling them separately has significantly broaden the application field of the existing symmetry breaking methods. The key question to address in the general resolution scheme was the search in canonical solution orbits. In the incomplete search context, we have seen that metaheuristic methods could apply easily. However the systematic search of an orbit requires attention and still a lot of work. Testing constraints of C_{rest} after generating a complete permutation would have been very inefficient. We have proposed a backtracking method that can reject a partial permutation before generating its completions. We have just mentioned a few ideas about filtering but gave no concrete algorithm about how to do so. How to adapt existing CSP mechanisms for maintaining local consistencies to the tree search of orbits needs to be investigated further.

We have also focused on the specific case of a symmetrical CSP with an asymmetrical cost function because it was a typical context where our solving methods could perform well. The experiments we conducted showed that our local or systematic methods could outperform the usual B&B method. The gain remains moderate but promising since our algorithms are still in a preliminary version and do not integrate filtering techniques for searching orbits.

References

[Benhamou, 1994] B. Benhamou. Study of symmetry in constraint satisfaction problems. In *Second Workshop on Principles and Practice of Constraint Programming (PPCP'94)*, 1994.

[Cohen *et al.*, 2005] D. Cohen, P. Jevons, C. Jefferson, K. E. Petrie, and B. Smith. Symmetry definitions for constraint satisfaction problems. In *Proceedings of CP'05*, pages 17–31, 2005.

[Harvey, 2005] W. Harvey. Symmetric relaxation techniques for constraint programming. In *SymNet Workshop on Almost-Symmetry in Search*, pages 20–59, 2005.

[Martin, 2005] R. Martin. Approaches to symmetry breaking for weak symmetries. In *SymNet Workshop on Almost-Symmetry in Search*, pages 37–49, 2005.

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

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

[Puget, 2005b] Jean Franois Puget. Breaking all value symmetries in surjection problems. In *Proceedings of CP'05*, pages 490–504, 2005.

[Seress, 1999] Ako Seress. *Permutation Group Algorithms*. Cambridge University Press, 1999.

Symmetry Breaking in Subgraph Pattern Matching

Stéphane Zampelli, Yves Deville, and Pierre Dupont

Université Catholique de Louvain,
Department of Computing Science and Engineering,
2, Place Sainte-Barbe
1348 Louvain-la-Neuve (Belgium)
{sz,yde,pdupont}@info.ucl.ac.be

Abstract

Graph pattern matching, a central application in many fields, can be modelled as a CSP. This CSP approach can be competitive with dedicated algorithms. In this paper, we develop symmetry breaking techniques for subgraph matching in order to increase the number of tractable instances. Specific detection techniques are first developed for the classical variables symmetries and value symmetries. It is also shown how these symmetries can be broken when solving subgraph matching. We also show how conditional value symmetries can be automatically detected and handled in the search process. Then, the concept of local value symmetries is introduced; it is shown how these symmetries can be computed and exploited. Finally, experimental results show that symmetry breaking is an effective way to increase the number of tractable instances of the subgraph matching problem.

1 Introduction

A symmetry in a Constraint Satisfaction Problem (CSP) is a bijective function that preserves CSP structure and solutions. Symmetries are important because they induce symmetric subtrees in the search tree. If the instance has no solution, failure has to be proved for equivalent subtrees regarding symmetries. If the instance has solutions, many symmetric solutions will have to be enumerated in symmetric subtrees. The detection and breaking of symmetries can thus speed up the solving of a CSP.

Symmetries arise naturally in graphs as automorphisms. However, although a lot of graph problems have been tackled [Beldiceanu *et al.*, 2005] [Cambazard and Bourreau, 2004] [Sellman, 2003] and a computation domain for graphs has been defined [Dooms *et al.*, 2005], and despite the fact that symmetries and graphs are related, little has been done to investigate the use of symmetry breaking for graph problems in constraint programming.

This paper aims at applying and extending symmetries techniques for subgraph matching. Existing techniques usually handle only initial symmetries and are not able to detect symmetries arising during search, so called conditional symmetries. We will show how to detect and handle those conditional symmetries.

Related Works Handling symmetries to reduce search space has been a subject of research in constraint programming for many years. Crawford and al. [Crawford *et al.*, 1996] showed that computing the set of predicates breaking the symmetries of an instance is NP-hard in general. Different approaches exist for exploiting symmetries. Symmetries can be broken during search either by posting additional constraints (SBDS) [Gent and Smith, 2001] [Gent *et al.*, 2002] or by pruning the tree below a state symmetrical to a previous one (SBDD) [Gent *et al.*, 2003]. Symmetries can be broken by taking into account the symmetries into the heuristic [Meseguer and Torras, 2001]. Symmetries can be broken by adding constraints to the initial problem at its root node [Crawford *et al.*, 1996] [Gent, 2001]. Symmetries can also be broken by remodelling the problem [Smith, 2001].

Dynamic detection of value symmetries and a general method for detecting them has been proposed in [Benhamou, 1994]. The general case for such a detection is difficult. However in not-equal binary CSPs some value symmetries can be detected in linear time [Benhamou, 2004] and dominance detection for value symmetries can be performed in linear time [Benhamou and Saïdi, 2006].

Lately research efforts has been triggered towards defining, detecting and breaking symmetries. Cohen and al. [Cohen *et al.*, 2005] defined two types of symmetries, solution symmetries and constraint symmetries and proved that the group of constraint symmetries is a subgroup of solution symmetries. Gent and al. [Gent *et al.*, 2005b] evaluated several techniques to break conditional symmetries, that is symmetries arising during search. However the detection of conditional symmetries remains a research topic. Symmetries were also shown to produce stronger forms of consistency and more efficient mechanisms for establishing them [Gent *et al.*, 2005a]. Finally, Puget [Puget, 2005b] showed how to detect symmetries automatically, and showed that all variable symmetries could be broken with a linear number of constraints for injective problems [Puget, 2005a].

Graph pattern matching is a central application in many fields [Conte *et al.*, 2004]. Many different types of algorithms have been proposed, ranging from general methods to specific algorithms for particular types of graphs. In constraint programming, several authors [Larrosa and Valiente, 2002;

Rudolf, 1998] have shown that subgraph matching can be formulated as a CSP problem, and argued that constraint programming could be a powerful tool to handle its combinatorial complexity. Within the CSP framework, a model for subgraph monomorphism has been proposed by Rudolf [Rudolf, 1998] and Valiente et al. [Larrosa and Valiente, 2002]. Our modeling [Zampelli *et al.*, 2005] is based on these works. Sorlin [Sorlin and Solnon, 2004] proposed a filtering algorithm based on paths for graph isomorphism and part of our approach can be seen as a generalization of this filtering. A declarative view of matching has also been proposed in [Mamoulis and Stergiou, 2004]. In [Zampelli *et al.*, 2005], we showed that CSP approach is competitive with dedicated algorithms over a graph database representing graphs with various topologies.

Objectives This work aims at developing symmetry breaking techniques for subgraph matching modelled as a CSP in order to increase the number of tractable instances of graph matching. Our first goal is to develop specific detection techniques for the classical variable symmetries and value symmetries, and to break such symmetries when solving subgraph matching. Our second goal is to develop more advanced symmetries that can be easily detected for subgraph matching.

Results

- We show that variable symmetries and value symmetries can be detected by computing the set of automorphisms on the pattern graph and on the target graph.
- We show that conditional value symmetries can be detected by computing the set of automorphisms on various subgraphs of the target graph, called dynamic target graphs. The GE-Tree method can be extended to handle these conditional symmetries.
- We introduce the concept of local value symmetries, that is symmetries on a subproblem. It is shown how such symmetries can be computed and exploited using standard methods such as GE-Tree.
- Experimental results compare and analyze the enhancement achieved by these symmetries and show that symmetry breaking is an effective way to increase the number of tractable instances of the subgraph matching problem.

Outline Sections 2 provides the necessary background in subgraph matching and in symmetry breaking. Section 3 describes a CSP approach for subgraph matching. Sections 3 and 4 present variable symmetries and value symmetries in subgraph matching. Conditional value symmetries are handled in Section 6, and Section 7 introduces local value symmetries in subgraph matching. Finally, Section 8 describes experimental results and Section 9 concludes this paper.

2 Background and Definitions

Basic definitions for subgraph matching and symmetries are introduced.

A **graph** $G = (N, E)$ consists of a **node set** N and an **edge set** $E \subseteq N \times N$, where an edge (u, v) is a pair of nodes. The nodes u and v are the endpoints of the edge (u, v). We consider directed and undirected graphs. A **subgraph** of a graph $G = (N, E)$ is a graph $S = (N', E')$ where N' is a subset of N and E' is a subset of E.

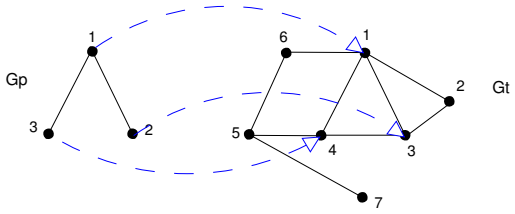


Figure 1: Example solution for a monomorphism problem instance.

A **subgraph monomorphism** (or subgraph matching) between G_p and G_t is a total injective function $f : N_p \rightarrow N_t$ respecting the monomorphism constraint : $(u, v) \in E_p \Rightarrow (f(u), f(v)) \in E_t$. Figure 1 shows an example of subgraph monomorphism.

The CSP model of subgraph matching should represent a total function $f : N_p \rightarrow N_t$. This total function can be modeled with $X = x_1, ..., x_n$ with x_i a FD variable corresponding to the ith node of G_p and $D(x_i) = N_t$. The injective condition is modeled with the global constraint $\texttt{alldiff}(x_1, ...x_n)$. The monomorphism condition is translated into the global constraint $\mathbb{MC}(x_1, ..., x_n) \equiv \bigwedge_{(i,j)\in E_p}(x_i, x_j) \in E_t$. Implementation, comparison with dedicated algorithms, and extension to subgraph isomorphism and to graph and function computation domains can be found in [Zampelli *et al.*, 2005; Deville *et al.*, 2005].

A CSP instance is a triple $< X, D, C >$ where X is the set of variables, D is the universal domain specifying the possible values for those variables, and C is the set of constraints. In the rest of this document, $n = |N_p|$, $d = |D|$, and $D(x_i)$ is the domain of x_i. A symmetry over a CSP instance P is a bijection σ mapping solutions to solutions, and hence non solutions to non solutions [Puget, 2005b]. Since a symmetry is a bijection where domain and target sets are the same, a symmetry is a permutation. A *variable symmetry* is a bijective function $\sigma : X \rightarrow X$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $s' = ((\sigma(x_1), d_1), \ldots, (\sigma(x_n), d_n))$. A *value symmetry* is a bijective function $\sigma : D \rightarrow D$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $s' = ((x_1, \sigma(d_1)), \ldots, (x_n, \sigma(d_n)))$. A *value and variable symmetry* is a bijective function $\sigma : X \times D \rightarrow X \times D$ permuting a (non) solution $s = ((x_1, d_1), \ldots, (x_n, d_n))$ to a (non) solution $s' = (\sigma(x_1, d_1), \ldots, \sigma(x_n, d_n))$. A *conditional symmetry* of a CSP P is a symmetry holding only in a sub-problem $P^{'}$ of P. The conditions of the symmetry are the constraints necessary to generate $P^{'}$ from P [Gent *et al.*, 2005b]. A *group* is a finite or infinite set of elements together with a binary operation (called the group operation) that satisfy the four fundamental properties of closure, associativity, the identity property, and the inverse property. An *automorphism of a graph* is a graph isomorphism with itself. The sets of automorphisms $Aut(G)$ define a finite permutation group.

Figure 2: Example of symbolic graph for a square pattern.

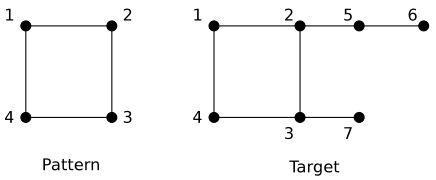


Figure 3: Example of matching where the set of value symmetries is not empty and $Aut(G_t) = \emptyset$.

3 Variable Symmetries

3.1 Detection

This section shows that, in subgraph matching, variable symmetries are the automorphisms of the pattern graph and do not depend on the target graph.

It has been shown that the set of variable symmetries of the CSP is the automorphism group of a *symbolic graph* [Puget, 2005b] . The pattern G_p is transformed into a symbolic graph $S(G_p)$ where $Aut(S(G_p))$ is the set of variable symmetries of the CSP.

Definition 1 *A CSP P modeling a subgraph monomorphism instance (G_p, G_t) can be transformed into the following symbolic graph $S(P)$:*

1. *Each variable x_i is a distinct node labelled i*
2. *If there exists a constraint $MC(x_i, x_j)$, then there exists an arc between i and j in the symbolic graph*
3. *The constraint alldiff is transformed into a node typed with label 'a'; an arc (a, x_i) is added to the symbolic graph.*

If we do not consider the extra node and arcs introduced by the alldiff constraint, then the symbolic graph $S(P)$ and G_p are isomorphic by construction. Given the labeling of nodes representing constraints, an automorphism in $S(P)$ maps the alldiff node to itself and the nodes corresponding to the variables to another node corresponding to the variables. Each automorphism in $Aut(G_p)$ will thus be a restriction of an automorphism in $Aut(S(P))$, and an element in $Aut(S(P))$ will be an extension of an element in $Aut(G_p)$. Hence the two following theorems.

Theorem 1 *Given a subgraph monomorphism instance (G_p, G_t) and its associated CSP P :*

- $\forall\, \sigma \in Aut(G_p)\ \exists\, \sigma^{'} \in Aut(S(P))$:
  $\forall\, n \in N_p$: $\sigma(n) = \sigma^{'}(n)$
- $\forall\, \sigma^{'} \in Aut(S(P))\ \exists\, \sigma \in Aut(G_p)$:
  $\forall\, n \in N_p$: $\sigma(n) = \sigma^{'}(n)$

Theorem 2 *Given a subgraph monomorphism instance (G_p, G_t) and its associated CSP P, the set of variable symmetries of P is the set of bijective functions $Aut(S(P))$ restricted to N_p, which is equal to $Aut(G_p)$.*

Theorem 2 says that only $Aut(G_p)$ has to be computed in order to get all variable symmetries.

Figure 2 shows a pattern transformed into its symbolic graph.

3.2 Breaking

Two techniques were selected to break variable symmetries. The first technique is an approximation and consists in breaking only the generators of symmetry group [Crawford *et al.*, 1996]. Those generators are obtained by using a tool such as NAUTY. For each generator σ, an ordering constraint $s \leq \sigma s$ is posted.

The second technique breaks all variable symmetries of an injective problem by using a SchreierSims algorithm, provided that the generators of the variable symmetry group are known [Puget, 2005b]. Puget showed the number of constraints to be posted is linear with the number of variables. The Schreier-Sims algorithm computes a base and strong generating set of a permutation group in $O(n^2 log^3|G| + t.n.log|G|)$, where G is the group, t the number of generators and n the size of the of group of all permutations containing G.

4 Value Symmetries

4.1 Detection

In subgraph matching, value symmetries are automorphisms of the target graph and do not depend on the pattern graph.

Theorem 3 *Given a subgraph monomorphism instance (G_p, G_t) and its associated CSP P, each $\sigma \in Aut(G_t)$ is a value symmetry of P.*

Proof Suppose $Sol = (v_1, \cdots, v_n)$ is a solution. Consider the subgraph $G = (N, E)$ of G_t, where $N = \{v_1, \cdots, v_n\}$ and $E = \{(i, j) \mid (\sigma^{-1}(i), \sigma^{-1}(j)) \in E_p\}$. This means there exists a monomorphic function $f^{'}$ matching G_p to σG. Hence $((x_1, \sigma(v_1)), \cdots, (x_n, \sigma(v_n)))$ is a solution. ■

All value symmetries of P are not in $Aut(G_t)$. Consider Figure 3. There exists two value symmetric solutions : $\{(x_1, 1), (x_2, 2), (x_3, 3), (x_4, 4)\}$ and $\{(x_1, 2), (x_2, 1), (x_3, 4), (x_4, 3)\}$ although $Aut(G_t) = \emptyset$.

4.2 Breaking

Breaking initial value symmetries can be done by using GE-Tree technique [C.M. *et al.*, 2004]. The idea is to modify the distribution by avoiding symmetrical value assignments. Suppose a state S is reached, where $x_1, \cdots, x_k$ are assigned to $v_1, \cdots, v_k$ respectively, and $x_{k+1}, \cdots, x_n$ are not assigned yet. The variable x_{k+1} should not be assigned to two symmetrical values, since two symmetric subtrees would be searched. For each value $v_i \in D(v_{k+1})$ that is symmetric to

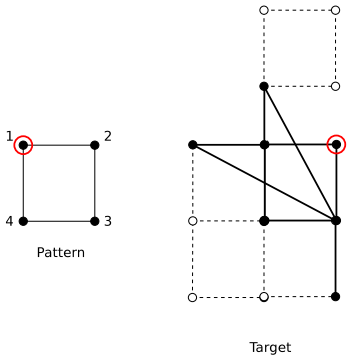


Figure 4: Example of dynamic target subgraph.

a value $v_j \in D(v_{k+1})$, only one state S_1 should be generated with the new constraint $x_{k+1} = v_i$; no new state S_2 with $x_i = v_j$ should be generated.

A convenient way to compute those symmetrical values is to compute a base and a strong generating set using the SchreierSims algorithm. Algorithm SchreierSims outputs the subgroups of $Aut(G_t)$ G_i $(1 \leq i \leq d)$ such that $\forall\ \sigma \in G_i : \sigma(j) = j\ \forall\ j \in [1, i]$ (called the pointwize stabilizators of G). Moreover SchreierSims outputs the set of images of i that let $0, \cdots, i$ invariant : $U_{i+1} = (i+1)^{G_{i+1}}$. Those sets U_i are interesting because they give the set of symmetrical values of i given that the values $1, ..., i$ are not subject to any permutation (mapped to themselves). If values are assigned in an increasing order, assigning symmetrical values can be avoided.

5 Conditional Value Symmetries

In subgraph monomorphism, the relations between values are explicitly represented in the target graph. This allows the detection of conditional values symmetries.

5.1 Detection

During the search, the target graph looses a node a whenever $a \notin \cup_{i \in N_p} D(x_i)$. This is interesting because the relation between the values are known dynamically.

The set of values $\cup_{i \in N_p} D(x_i)$ denotes the nodes of subgraph of G_t in which a solution is searched. For a given state S, such a subgraph can be, for a given state S, computed efficiently. We first define this subgraph of G_t.

Definition 2 *Let S be a state in the search where $x_1, \cdots, x_k$ are assigned, and $x_{k+1}, \cdots, x_n$ are not assigned. The* ***dynamic target graph*** *$G_t^* = (N_t^*, E_t^*)$ is a subgraph of G_t such that :*

- $N_t^* = \cup_{i \in [1, \cdots, n]} D(x_i)$
- $E_t^* = \{(a, b) \in E_t \mid a \in N_t^* \wedge b \in N_t^*\}$

Figure 4 shows an example of dynamic target graph. In this figure, the circled nodes are assigned together. The blank nodes are the nodes excluded from $\cup_{i \in [1, \cdots, n]} D(x_i)$, and the black nodes are the nodes included in $\cup_{i \in [1, \cdots, n]} D(x_i)$. The plain edges are the selected edges for the dynamic target subgraph.

Each automorphism of G_t^* is a conditional value symmetry for the state S.

Theorem 4 *Given a subgraph monomorphism instance (G_p, G_t), its associated CSP P, and a state S in the search, each $\sigma \in Aut(G_t^*)$ is a conditional value symmetry of P. Moreover, the conditions of σ are $x_1 = v_1, \cdots, x_k = v_k$.*

Proof Suppose $Sol = (v_1, \cdots, v_k)$ is a partial solution. Consider the subgraph G_t^*. The state S can be considered as a new CSP $P^{'}$ of an instance (G_p, G_t^*) with additional constraints $x_1 = v_1, \cdots, x_k = v_k$. By Theorem 3, the thesis follows. ■

The size of G_t^* is an important issue, as we will dynamically compute symmetry information with it. The following theorem shows that, because of the MC constraints, the longest path in G_p has the same length than the longest path in G_t whenever at least a variable is assigned.

Definition 3 *Let $G = (N, E)$ be a graph. Then $maxd(G)$ denotes the size of the longest simple path between two nodes $a, b \in N$.*

Theorem 5 *Given a subgraph monomorphism instance (G_p, G_t), its associated CSP P, and a state S in the search, if $\exists\ i \in N_p \mid |D(x_i)| = 1$, then $maxd(G_p) = maxd(G_t^*)$.*

This is a nice result for complexity issues, when $maxd(G_p)$ is small. In Figure 4, $maxd(G_p)$=2 and only nodes at shortest distance 2 from node 1 in the target graph are included in G_t^*.

The dynamic target graph G can be computed dynamically. In [Deville *et al.*, 2005], we showed how subgraph matching can be modelled and implemented in CP(Graph), an extension of CP with graph domain variables. In this setting, a graph domain variable T is used for target graph, with initial domain $[\emptyset, \cdots, G_t]$. When a solution is found, T is instantiated to the matched subgraph of G_t. Hence, during the search, the dynamic target graph G_t^* will be the upper bound of variable T and can be obtained in $O(1)$.

5.2 Breaking

In this subsection, we show how to modify GE-Tree method to handle conditional value symmetries. Before distribution, the following actions are triggered :

1. Get G_t^*.
2. The NAUTY and SchreierSims algorithms are called. This returns the new $U_i^{'}$ sets.
3. The main problem is how to adapt the variable and value selection such that conditional value symmetries are broken. In GE-Tree, from a given state S, two branches are created :
   (a) a new state S_1 with a constraint $x_k = v_k$
   (b) a new state S_2 with constraints :
      i. $x_k \neq v_k$
      ii. $x_k \neq v_j\ \forall\ j \in U_{k-1}$.

   To handle conditional value symmetries, we slightly modify this schema. From a given state S, two branches are created :

(a) a new state S_1 with a constraint $x_k = v_k$

(b) a new state S_2 with constraints :

i. $x_k \neq v_k$

ii. $x_k \neq v_j \; \forall \; j \in U_{k-1} \cup U'_{k-1}$

An issue is how to handle structure U. In Gecode system (http://www.gecode.org), in which the actual implementation is made, the states are copied and trailing is not needed. Thus the structure U must not be updated because of backtracking. A single global copy is kept during the whole search process. In a state S where conditional values symmetries are discovered, structure U is copied into a new structure $U^{''}$ and merged with $U^{'}$. This structure $U^{''}$ shall be used for all states $S^{'}$ having S in its predecessors. Of course, some heuristics should be added to choose the states where a new conditional value symmetry should be computed.

6 Local Value Symmetries

In this section, we introduce the concept of local value symmetries, that is value symmetries on a subproblem. This concept can be seen as a particular case of dynamic detection of value symmetries such as studied in [Benhamou, 1994]. However local values symmetries exploits the fact that in subgraph monomorphism relations between values are explicitly represented in the target graph.

6.1 Detection

We first introduce partial dynamic graphs. Those graphs are associated to a state in the search and correspond to the unsolved part of the problem. This can be viewed as a new local problem to the current state.

Definition 4 *Let S be a state in the search whose variables $x_1, \cdots, x_k$ are assigned to $v_1, \cdots, v_k$ respectively, and $x_{k+1}, \cdots, x_n$ are not assigned yet.*
*The **partial dynamic pattern graph** $G_p^- = (N_p^-, E_p^-)$ is a subgraph of G_p such that :*

- $N_p^- = \{i \in [k+1, n]\}$
- $E_p^- = \{(i,j) \in E_p \mid i \in N_p^- \wedge j \in N_p^-\}$

*The **partial dynamic target graph** $G_t^- = (N_t^-, E_t^-)$ is a subgraph of G_t such that :*

- $N_t^- = \cup_{i \in [k+1,n]} D(x_i)$
- $E_t^- = \{(a,b) \in E_t \mid a \in N_t^- \wedge b \in N_t^-\}$

When forward checking (FC) is used during the search, in any state in the search tree, every constraint involving *one* uninstantiated variable is arc consistent. In other words, every value in the domain of an uninstantiated variable is consistent with the partial solution. This FC property on a binary CSP ensures that one can focus on the uninstantiated variables and their associated constraints without losing or creating solutions to the initial problem. Such a property also holds when the search achieves stronger consistency in the search tree (Partial Look Ahead, Maintaining Arc Consistency, ...).

Theorem 6 *Let (G_p, G_t) be a subgraph monomorphism instance, P its associated CSP, and S a state of P during the search, where the assigned variables are $x_1, \cdots, x_k$ with values $v_1, \cdots, v_k$. Let P' be a new CSP of a subgraph monomorphism instance (G_p^-, G_t^-) with additional constraints $x'_{k+1} = D(x_{k+1}), \cdots, x'_n = D(x_n)$. Then:*

1. *Each $\sigma \in Aut(G_t^-)$ is a value symmetry of P'.*
2. *Assuming we have the FC property, we have $((x_1, v_1), \cdots, (x_n, v_n)) \in Sol(S)$ iff $((x_{k+1}, v_{k+1}), \cdots, (x_n, v_n)) \in Sol(P')$.*

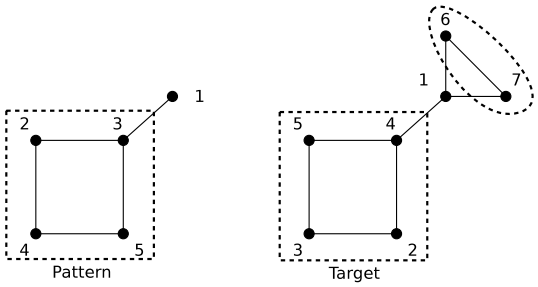


Figure 5: Example of conditional local value symmetry. The dashed lines show the new subgraph monomorphism instance for CSP P'.

The theorem states that value symmetries of the local CSP P' can be obtained by computing $Aut(G_t^-)$ and that these symmetries can be exploited without losing or adding solutions to the initial matching problem.

It is important to notice that the value symmetries of P' are *not* conditional symmetries of P. It is not possible to add constraints to P to generate P'. As the CSP P' is a local CSP associated to a state S, these value symmetries are called *local value symmetries*.

The computation of G_t^- can be easily performed thanks to graph variables. If T is the target graph variable over initial domain $[\emptyset, \cdots, G_t]$, then during the computation G_t^- is $lub(T) \setminus glb(T)$.

Consider the subgraph monomorphism instance (G_p, G_t) in Figure 5. Nodes of the pattern graph are the variables of the corresponding CSP, i.e. node i of G_p corresponds to variable x_i. Suppose that x_1 has been assigned to value 1. Because of MC(x_1, x_3), $D(x_3) = \{4, 6, 7\}$. Moreover, because of alldiff($x_1, \cdots, x_n$), value 1 is deleted from all domains $D(x_i)$ $(i \neq 1)$. The new CSP P' consists of the subgraph of $G_p^- = (\{2, 3, 4, 5\}, \{(2,3), (3,2), (3,5), (5,3), (4,5), (5,4), (2,4), (4,2)\})$ and $G_t^- = (\{2, 3, 4, 5, 6, 7\}, \{(2,3), (3,2), (3,5), (5,3), (4,5), (5,4), (2,4), (4,2), (7,6), (6,7)\})$. The domains of the variables of P' are : $D(x_3) = \{4, 6, 7\}$, $D(x_2) = \{2, 5, 6, 7\}$, $D(x_5) = \{2, 5, 6, 7\}$, $D(x_4) = \{3, 4, 6, 7\}$. For the state S, $Sol(S) = \{(1, 5, 4, 3, 2), (1, 2, 4, 3, 5)\}$ and $BSol(S) = \{(1, 2, 4, 3, 5)\}$. For the subproblem P', $Sol(P') = \{(5, 4, 3, 2), (2, 4, 3, 5)\}$ and $BSol(P') = \{(2, 4, 3, 5)\}$. The partial assignment $(x_1, 1)$ in state S together with the solutions of P' equals $Sol(S)$.

6.2 Breaking

Breaking local value symmetries is equivalent to breaking value symmetries on the subproblem P'. Puget's method and the dynamic GE-Tree method can thus be applied to the local CSP P'.

7 Experimental results

The CSP model for subgraph monomorphism has been implemented in Gecode (http://www.gecode.org), using CP(Graph) and CP(Map) [Dooms *et al.*, 2005] [Deville *et al.*, 2005] . CP(Graph) provides graph domain variables and CP(Map) provides function domain variables. All the software was implemented in C++. The standard implementation of NAUTY algorithm was used. We also implemented SchreierSims algorithm. The computation of the constraints for breaking injective problems was implemented, and GE-Tree method was also incorporated.

We have evaluated variable symmetry detection and breaking, value symmetry detection and breaking, and variable and value symmetry breaking.

The data graphs used to generate instances are from the GraphBase database containing different topologies and has been used in [Larrosa and Valiente, 2002]. There is a directed and an undirected set of graphs. We took the first 30 graphs and the first 50 graphs from GraphBase. The directed set contains graphs ranging from 10 nodes to 462 nodes. The undirected set contains graphs ranging from 10 nodes to 138 nodes. Using those graphs, there are 405 instances for directed graphs and 1225 instances for undirected graphs. All runs were performed on a dual Intel(R) Xeon(TM) CPU 2.66GHz with 2 Go of RAM.

A main concern is how much time it takes to preprocess the graphs. NAUTY processed each undirected graph in less than 0.02 second. For directed graphs, each graph was processed in less than 0.01 second except one of them which terminate in 0.8 second and 4 of them which did not terminate in five minutes. Note that we did not tune NAUTY. For the SchreierSims algorithm, each directed graph was processed in less than one second except for 3 of them which terminate in 0.5 second, 1 of them in 1.5 seconds, and 1 of them in 3.1 seconds. All undirected graphs were processed in less than one second, except two of them, with 4 seconds and 8 seconds.

In our tests, we look for all solutions. A run is solved if it finishes under 5 minutes, unsolved otherwise. We applied the basic CSP model, the model where breaking variable symmetries with generators (Gen.) are posted, and finally the full variable symmetry (FVS) that breaks all variable symmetries. Results are shown in Table 1 and 2. In those runs, the preprocessing time has not been considered. The total time column shows the total time needed for the solved instances. The mean time column shows the mean time for the solved instances.

Thanks to variable symmetry breaking constraints more instances are solved, either for the directed graphs or for the undirected graphs. Moreover, the time for solved instances was increased because of the variable symmetry breaking constraints. Regarding the mean time, the full variable symmetry breaking constraint has a clear advantage. This mean time increase is an astonishing behavior that should be investigated.

Table 1: Comparison over GraphBase undirected graphs.

All solutions 5 min.				
	solved	unsol	total time	mean time
CSP	58%	42%	70 min.	5.95 sec.
Gen.	60,5%	39,5%	172 min.	13.95 sec.
FVS	61.8%	38.2%	101 min.	8 sec.

Table 2: Comparison over GraphBase directed graphs.

All solutions 5 min.				
	solved	unsol	total time	mean time
CSP	67%	33%	21 min.	4.31 sec.
Gen.	74%	26%	47 min.	8.87 sec.
FVS	74%	26%	40 min.	7.64 sec.

Value symmetry breaking was evaluated on the set of directed graphs. Table 3 shows that only one percent was gained. This may be due to the fact that there are less symmetries in directed graph than in undirected graphs. For variable and value symmetries, a total of 233 undirected random instances were treated. We evaluated variable and values symmetries separately and then together in Table 4. This table shows that, as expected, value symmetries and variable symmetries each increase the number of solved instances. Notice here that value symmetry breaking with GE-Tree leads to new solved instances and better performance, reducing mean time on solved instances. Full variable symmetry technique makes new instances solved, but does not significantly reduce mean time on solved instances. Moreover, the combination of value symmetry breaking and variable symmetry breaking do not combine the power of the two techniques. In fact the GE-Tree upper bound of the number of the solved solutions is not increased by using full variable symmetry technique, and its mean time is even increased.

From these experiments, we conclude that although variable and value symmetry gives better performances and make new instances solved, they are not sufficient to make a significant higher percentage of instances solved. This calls for conditional and local symmetry detection and breaking.

8 Conclusion

In this paper, we presented techniques for symmetry breaking in subgraph matching. Specific detection techniques were first developed for the classical variables symmetries and value symmetries. We show that variable symmetries and value symmetries can be detected by computing the set of automorphisms on the pattern graph and on the target graph. We also showed that conditional value symmetries can be detected by computing the set of automorphisms on various subgraphs of the target graph, called dynamic target graphs.

Table 3: Comparison over GraphBase directed graphs for value symmetries.

All solutions 5 min.				
	solved	unsol	total time	mean time
GE-Tree	68%	32%	21 min.	4.39 sec.

Table 4: Comparison over GraphBase undirected graphs for variable and value symmetries.

All solutions 5 min.				
	solved	unsol	total time	mean time
CSP	53,6%	46,3 %	31 min.	20.1 sec.
GE-Tree	55,3%	44,7 %	6 min.	3.21 sec.
FVS	54,9 %	45,1%	31 min.	19 sec.
GE-Tree and FVS	55,3 %	44,7%	26 min.	8.68 sec.

The GE-Tree method has been extended to handle these conditional symmetries. We introduced the concept of local value symmetries, that is symmetries on a subproblem. It was shown how such symmetries can be computed and exploited using standard methods such as GE-Tree. Experimental results analyzed the enhancement achieved by variables symmetries and value symmetries. It showed that symmetry breaking is an effective way to increase the number of tractable instances of the graph matching problem.

Future work includes more experiments on conditional symmetries and local value symmetries, and the development of heuristics for the integration of these symmetries on suitable search states. An interesting research direction is the automatic detection of symmetries in graph domain variable. Finally, an open issue is the ability to handle local variable symmetries.

References

[Beldiceanu *et al.*, 2005] N. Beldiceanu, P. Flener, and X. Lorca. The tree constraint. In *Proceedings of CP-AI-OR'05*, volume LNCS 3524. Springer-Verlag, 2005.

[Benhamou and Saïdi, 2006] Belaïd Benhamou and Mohamed Réda Saïdi. Reasoning by dominance in not-equals binary constraint networks. In LNCS Springer, editor, *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP-2006)*, Cité des Congrès - Nantes, France, septembre 2006. to appear.

[Benhamou, 1994] Belaid Benhamou. Study of symmetry in constraint satifaction. In *PCP'94*, 1994.

[Benhamou, 2004] Belaid Benhamou. Symmetry in not-equals binary constraint networks. In *Proceedings of the satelite workshop of CP 2004, Symmetry in Constraints (SymCon'04)*, pages 2–8, september 2004.

[Cambazard and Bourreau, 2004] H. Cambazard and E. Bourreau. Conception d'une constrainte globale de chemin. In *10e Journ. nat. sur la résolution de problèmes NP-complets (JNPC'04)*, pages 107–121, 2004.

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

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

[Conte *et al.*, 2004] Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18(3):265–298, 2004.

[Crawford *et al.*, 1996] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry breaking predicates for search problem. In *Proceedings of KR'96*, 1996.

[Deville *et al.*, 2005] Yves Deville, Grégoire Dooms, Stéphane Zampelli, and Pierre Dupont. Cp(graph+map) for approximate graph matching. *1st International Workshop on Constraint Programming Beyond Finite Integer Domains, CP2005*, 2005.

[Dooms *et al.*, 2005] Grégoire Dooms, Yves Deville, and Pierre Dupont. Cp(graph): Introducing a graph computation domain in constraint programming. *Principles and Pratice of Constraint Programming*, 2005.

[Gent and Smith, 2001] I.P. Gent and B.M. Smith. Symmetry breaking during search in constraint programming. In *Proceedings of CP'01*, pages 599–603, 2001.

[Gent *et al.*, 2002] I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints : symmetry breaking during search. In *Proceedings of CP'02*, pages 415–430, 2002.

[Gent *et al.*, 2003] I.P. Gent, W. Harvey, and T. Kelsey. Generic sbdd using computational group theory. In *Proceedings of CP'03*, pages 333–346, 2003.

[Gent *et al.*, 2005a] Ian .P. Gent, Tom Kelsey, Steve Linton, and Colva Roney-Dougal. Symmetry and consistency. In van Beek [2005], pages 271–285.

[Gent *et al.*, 2005b] Ian .P. Gent, Tom Kelsey, Steve A. Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith. Conditional symmetry breaking. In van Beek [2005], pages 256–270.

[Gent, 2001] I.P. Gent. A symmetry breaking constraint for indistinguishable values. In *Proceedings of CP'01, SymCon'01 Workshop*, 2001.

[Larrosa and Valiente, 2002] Javier Larrosa and Gabriel Valiente. Constraint satisfaction algorithms for graph pattern matching. *Mathematical. Structures in Comp. Sci.*, 12(4):403–422, 2002.

[Mamoulis and Stergiou, 2004] Nikos Mamoulis and Kostas Stergiou. Constraint satisfaction in semi-structured data graphs. In Mark Wallace, editor, *CP2004*, volume 3258 of *Lecture Notes in Computer Science*, pages 393–407. Springer, 2004.

[Meseguer and Torras, 2001] P. Meseguer and C. Torras. Exploiting symmetries within the constraint satisfaction search. *Artificial intelligence*, 129(1-2):133–163, 2001.

[Puget, 2005a] Jean-Francois Puget. Elimination des symétries dans les problèmes injectifs. In *Proceedings des Journées Francophones de la Programmation par Contraintes*, 2005.

[Puget, 2005b] Jean-François Puget. Automatic detection of variable and value symmetries. In van Beek [2005], pages 477–489.

[Rudolf, 1998] Michael Rudolf. Utilizing constraint satisfaction techniques for efficient graph pattern matching. In

Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *TAGT*, volume 1764 of *Lecture Notes in Computer Science*, pages 238–251. Springer, 1998.

[Sellman, 2003] M. Sellman. Cost-based filtering for shorter path constraints. In *Proc. of the 9th International Conference on Principles and Pratice of Constraint Programming (CP).*, volume LNCS 2833, pages 694–708. Springer-Verlag, 2003.

[Smith, 2001] B. Smith. Reducing symmetry in a combinatorial design problem. *Proc. CP-AI-OR'01, 3rd Int. Workshop on Integration of AI and OR Techniques in CP*, 2001.

[Sorlin and Solnon, 2004] Sébastien Sorlin and Christine Solnon. A global constraint for graph isomorphism problems. In Jean-Charles Régin and Michel Rueher, editors, *CPAIOR*, volume 3011 of *Lecture Notes in Computer Science*, pages 287–302. Springer, 2004.

[van Beek, 2005] Peter van Beek, editor. *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, Augustus 1-5, 2005, Proceedings*, volume 3709 of *Lecture Notes in Computer Science*. Springer, 2005.

[Zampelli *et al.*, 2005] Stéphane Zampelli, Yves Deville, and Pierre Dupont. Approximate constrained subgraph matching. *Principles and Pratice of Constraint Programming*, 2005.

Dynamic Symmetry Breaking Restarted

Daniel S. Heller and Meinolf Sellmann
Brown University, Department of Computer Science
P.O. Box 1910, Providence, RI 02912, U.S.A.
dheller,sello@cs.brown.edu

Abstract

Recently, structural symmetry breaking (SSB), a new technique for breaking all piecewise variable and value symmetry in constraint satisfaction problems (CSPs), was introduced. As of today, it is unclear whether the heavy symmetry filtering that SSB performs is at all worthwhile. This paper has two aims: First, we assess the feasibility of SSB. To this end, we introduce the first random benchmark generator that produces CSP instances with piecewise symmetric variables and values of constrainedness. It allows us to evaluate SSB on different regions of constrainedness. Secondly, we study how symmetry breaking and restarts interact. We propose practical enhancements of SSB that allow us to re-use symmetry no-goods in subsequent restarts efficiently. With those enhancements, we find that symmetry breaking can actually benefit from restarts. However, the improvements to be gained by restarting are far smaller than those that can be obtained for methods that break only some symmetries or none at all. Surprisingly, we find that a combination of restarts and breaking value symmetry only can be competitive with, or even be superior to, complete symmetry breaking.

Keywords: structural symmetry breaking, dynamic symmetry breaking, piecewise symmetry

1 Introduction

Symmetry breaking has received considerable and increasing interest in past years. It is widely accepted now that symmetries can cause significant problems to systematic solvers that unnecessarily explore redundant parts of the search tree. Methods to avoid this undesirable behavior range from adapting ordering heuristics [2], adding static constraints to the problem [4; 6], adding constraints during search [9], and filtering values based on a symmetric dominance analysis when comparing the current search node with those that were previously expanded [5; 7; 1; 13].

Especially the latter technique, known as symmetry breaking by dominance detection (SBDD), has proven to excel on problems that contain large symmetry groups. The core task of SBDD is the dominance detection algorithm. The first automated dominance detection algorithms were based on group theory [8], while the first provably polynomial-time dominance checkers for specific types of value symmetry were devised in [16]. This work was later extended to tackle any kind of value symmetry in polynomial time [14]. Based on these results, for specific "piecewise" symmetric problems, [15] showed that breaking variable and value symmetry can be broken simultaneously in polynomial time. The method was named structural symmetry breaking (SSB) and is based on the structural abstraction of a given partial assignment of values to variables.

Compared with other symmetry breaking techniques, the big advantage of dynamic symmetry breaking is that it can accommodate dynamic variable and value orderings. Dynamic orderings have shown to be vastly superior to static orderings in many different types of constraint satisfaction problems. However, robust heuristics for the selection of variables and values are hard to come by. For the task of variable selection, a bias towards variables with smaller domains often works comparably well, but there always remains a fair probability that we hit instances on which a solver gets trapped in extremely long runs. Particularly, heavy-tailed runtime distributions have been reported [10]. One way to circumvent this problematic situation is to randomize the solver and to restart the search when a run takes too long [11]. While dynamic symmetry breaking and restarts are orthogonal techniques in that they can be applied independently from one another, their interplay has not been studied yet.

In this contribution, we wish to investigate questions regarding dynamic symmetry breaking in general and SSB in particular. While the existing theoretical worst-case analysis of SSB shows that we can guarantee symmetry-free search trees in polynomial time, it remains unclear so far whether we can also implement the method so that it performs well in practice. We introduce practical enhancements of SSB such as delayed ancestor-based filtering and incremental data structures for sibling-based filtering. We then apply the method in combination with one-shot and restarted solvers. To conduct this study, we introduce the first randomized test suite for symmetry breaking experiments. It allows us to evaluate the method over an entire region of constrainedness rather than on isolated benchmark instances of unknown level of constrainedness only.

The paper is organized as follows: In the following section, we briefly review structural symmetry breaking. We discuss how symmetry breaking by dominance detection can be ex-

ploited as a no-good store between restarts in Section 3. Then, in Sections 4 and 5 we devise efficient mechanisms to speed-up structural symmetry breaking in practice by introducing delayed ancestor-based filtering and by devising an incremental data structure for sibling-based filtering. The technical core of the paper concludes in Section 6 where we present extensive numerical results on the effect of our enhancements as well as the interplay of symmetry breaking and restarts.

2 Background

Recently, a new technique was developed that, for the first time, allows us to simultaneously break value and variable symmetry in CSPs [15]. This technique, named *structural symmetry breaking (SSB)*, is based on the quantitative abstraction of a constraint program that contains sets of pairwise symmetric variables and values. Before we study implementation issues later, we now give a high-level description of SSB.

Definition 1

- *A* Constraint Satisfaction Problem (CSP) *is a tuple* (Z, V, D, C) *where* $Z = \{X_1, \ldots, X_n\}$ *is a finite set of variables,* $V = \{v_1, \ldots, v_m\}$ *is a set of values,* $D = \{D_1, \ldots, D_n\}$ *is a set of finite domains where each* $D_i \in D$ *is the set of possible instantiations to variable* X_i*, and* $C = \{c_1, \ldots, c_p\}$ *is a finite set of constraints where each* $c_i \in C$ *is defined on a subset of the variables in* Z *and specifying their valid combinations. We say that the CSP has* scalar variables *iff for all* $D_i \in D$ *it holds that* $D_i \subseteq V$*. Throughout this paper, we will consider scalar CSPs.*
- *Given a CSP with scalar variables, an* assignment A *is a set of pairs* $(X, v) \in Z \times V$ *such that* $(X, v), (X, w) \in A$ *implies* $v = w$*. An assignment of cardinality* n *is called* complete*, otherwise it is called* partial*. A complete assignment satisfying all constraints is called a* solution*.*

Definition 2

- *Given a set* S *and a set of sets* $P = \{P_1, \ldots, P_r\}$ *such that* $\bigcup_i P_i = S$ *and the* P_i *are pairwise non-overlapping, we say that* P *is a* partition *of* S*, and we write* $S = \sum_i P_i$*.*
- *Given a set* S *and a partition* $S = \sum_i P_i$*, a bijection* $\pi : S \mapsto S$ *such that* $\pi(P_i) = P_i$ *(where* $\pi(P_i) = \{\pi(s) \mid s \in P_i\}$*) is called a* piecewise permutation *over* $S = \sum_i P_i$*.*

The type of symmetry that the method can tackle efficiently is defined as follows:

Definition 3

- *Given a CSP* (Z, V, D, C)*, and partitions* $Z = \sum_{k \leq r} P_k$*,* $V = \sum_{l \leq s} Q_l$*, we say that the CSP has* piecewise variable and value symmetry *iff all variables within each* P_k *and all values within each* Q_l *are considered as interchangeable [3].*
- *Given two assignments* A *and* B *on a piecewise symmetric CSP, we say that* A dominates B *iff there exist piecewise permutations* π *over* $Z = \sum_{k \leq r} P_k$ *and* α *over* $V = \sum_{l \leq s} Q_l$ *such that for all* $(X, v) \in A$ *it holds that* $(\pi(X), \alpha(v)) \in B$*.*
- *Given two arbitrary assignments* A *and* B *for a piecewise symmetric CSP, we call the problem of determining whether* A *dominates* B *the* Dominance Detection Problem*.*

2.1 SSB for Dominance Checking

The core idea to devising an efficient dominance checker for piecewise symmetric CSPs lies in the definition of *signatures* of values under an assignment.

Definition 4

- *Given a partial assignment* A*, for all values* v*, we define*

$$sign_A(v) := (|\{X_i \in P_k \mid (X_i, v) \in A\}|)_{k \leq r},$$

where k *indexes the different variable partitions* $\sum_{k \leq r} P_k$*. That is, the signature of* v *under* A *is the tuple that counts, for each variable partition, by* how many *variables in the partition the value is taken in* A*.*
- *We say that a value* v *in an assignment* A dominates *a value* w *in assignment* B *iff* v *and* w *belong to the same value-symmetry class and* $sign_A(v) \leq sign_B(w)$*.*[1]
- *We say that a value* v *in an assignment* A *is* structurally equivalent *to a value* w *in assignment* B *iff* v *and* w *belong to the same value-symmetry class and* $sign_A(v) = sign_B(w)$*.*

The following result, from [15], connects dominance relations and partial assignments.

Lemma 1
An assignment A *dominates another assignment* B *in a piecewise symmetric CSP iff there exists a piecewise permutation* α *over* $\sum_{l \leq s} Q_l$ *such that* v *in* A *dominates* $\alpha(v)$ *in* B *for all* $v \in V$*.*

This lemma allows us to check dominance between assignments A and B: We set up a bipartite graph where, for each value v, there is one node on the left and one on the right. An edge connects two nodes with associated values v and w from the same value partition iff $sign_A(v) \leq sign_B(w)$. Then, A dominates B iff the bipartite graph contains a perfect matching.

2.2 SSB for Filtering

Based on the dominance checking algorithm, we can now filter values from domains iff setting the respective variable to some value would lead to a symmetric choice point. Since symmetry-based filtering *anticipates* when variable assignments will result in symmetric configurations, within SSB we have to distinguish two different types of filtering: *ancestor-based filtering* where we compare extensions to the current partial assignment with previously fully expanded search nodes, and *sibling-based filtering* where we compare extensions to the current partial assignments with other such extensions.

[1] Where the $\leq$-relation on vectors is defined as the usual component-wise comparison, i.e.: $x \leq y$ iff $x_i \leq y_i \; \forall \; i$.

The latter is very easy to handle as sibling-symmetry can only be caused by value symmetry in the problem.[2] Consequently, we can break all sibling symmetry simply by choosing only one arbitrary value out of each group of values within the same value partition that have the same signature. Ancestor-based filtering on the other hand can be performed by considering almost successful dominance checks: When the bipartite graph that we set-up contains an almost perfect matching where just one more edge is missing to complete it, we can quickly identify such *critical edges* and check whether one more variable assignment would cause the critical edge to be added to the graph. In this case, we have found a *critical variable assignment* that can and should be avoided by removing the respective value from the variable's domain.

This concludes our brief review of SSB. For a more detailed description of the method and a worst-case asymptotic runtime analysis, we refer the reader to [15].

3 Symmetry No-goods and Restarts

Branching decisions are critical to the efficiency of systematic search, and truly robust heuristics for choosing branching variables are not known. Empirical analysis of runtime distributions with randomized choices of branching variables has revealed that there is a substantial chance of very long runs [10]. At the same time, there is also a good probability that a random selection of variables will result in a short run. Consequently, it has been suggested to simply restart backtracking solvers when a run takes too long. This method of restarted randomized searches has been one key element of the latest generations of outstandingly powerful DPLL-based SAT-solvers.

Restarts work already quite well when everything is forgotten between two runs. In SAT, however, no-good learning algorithms augment the SAT-formula during search so as to improve the performance of unit propagation within DPLL. The no-goods learned implicitly store information on which parts of the search space have already been investigated. Moreover, they also contain information on which parts have not been investigated yet but cannot contain solutions as search would fail for the same reasons as it did earlier. Within one run of a backtracking solver, the first information is obviously obsolete as the systematic search already guarantees that the same part of the search space will not be investigated twice. However, the information becomes interesting when restarts are being used.

With respect to symmetry breaking, SSB (as a special form of SBDD) stores the most general previously fully expanded search nodes as a list of no-goods. In contrast to ordinary no-goods, an SBDD no-good implicitly represents an equivalence class of no-goods (namely the set of all its symmetric variants), and it is the algorithmic task of the dominance checker to see whether this set contains a no-good that is relevant with respect to the current search node. In this view, SBDD resembles the task of performing inference in predicate logic where we also need the help of a unification algorithm to find an applicable rule or fact as represented implicitly by all-quantified rules and facts in the formula.

[2] This assumes that all siblings are generated by branching on the same variable.

What is interesting to note is that SBDD no-goods also keep a record of those parts of the search space that have already been searched through. In that regard, it is of interest to store them (or at least the most powerful ones) between restarts. There is a trade-off, however: No-goods will only be beneficial if the method that prevents us from exploring the same part of the search space more than once does not impose a greater computational cost than what the exploration would cost anyway. One simple thing that we can do is to remove those no-goods from the list that have very little impact anyway because they only represent a small part of the search space. This is an idea that is commonly used in SAT. However, for symmetry-nogoods we can do more.

4 Delayed Ancestor-based Filtering

We introduce delayed symmetry filtering. The core idea here is to apply an inexpensive inference mechanism that quickly identifies which no-goods cannot cause effective symmetry-based filtering at a given search node. The aim here is to save many of the expensive calls to SSB-based domain filtering as described in the previous section. Note that no-goods are only used for ancestor-based filtering, which is why this idea will only be applied for this type of symmetry filtering. We discuss special methods to improve the performance of sibling-based filtering in Section 5.

4.1 A Simple Pretest

To cut down on full-fledged ancestor-based filtering calls, we introduce a simple pretest. What we need to identify are simple conditions under which a previously expanded node α (as usual, α is identified with the partial assignment that leads to the node) cannot "almost dominate" the current search node β. Precisely, α "almost dominates" β if one more assignment to β could result in a successful dominance relation with α. This is a necessary condition for SSB filtering to have any effect

First, we observe that β must contain exactly one less variable assignments as α. This is a trivial condition which is always true in a one-shot tree search as all no-goods stored by SBDD were taken from search nodes at the same or lower depth as that of the current node. However, for no-goods stored in earlier restarts, this test can quickly reveal that ancestor-based filtering will not be effective.

Only if the above condition holds, we perform one more test before applying the full-fledged filtering call: we look more closely at the two assignments α and β and see whether α is close to dominating β. Before looking at each value individually, determining all their signatures and which ones dominates which, we can do the same on the level of value classes: For each value partition, we determine how many variables in each value partition are taking a value in it under assignment γ, thus computing a signature for each partition of mutually symmetric values:

$$sign_\gamma(Q_l) := (|\{X \in P_k \mid \gamma(v) \in Q_l\}|)_{k \leq r} \qquad \forall\, 1 \leq l \leq s.$$

Now, from the description of SSB dominance checking in Section 2.1, we can infer:

Lemma 2
Given assignments α and β such that α dominates β, we have that, for all $1 \leq l \leq s$, $sign_\alpha(Q_l) \leq sign_\beta(Q_l)$ (whereby $\leq$ denotes the component-wise comparison of the two tuples).

Proof: Let $l \in \{1, \ldots, s\}$. Since α dominates β, we have that, for all $v \in Q_l$, $sign_\alpha(v) \leq sign_\beta(w)$ for some value $w \in Q_l$ that is the unique matching partner of v. Consequently,

$$sign_\alpha(Q_l) = \sum_{v \in Q_l} sign_\alpha(v) \leq \sum_{w \in Q_l} sign_\beta(w) = sign_\beta(Q_l).$$

■

Thus, SSB filtering can only be effective if the inequality holds for all but at most one value partition l, and if for that partition we have that $sign_\alpha(Q_l) \leq sign_\beta(Q_l) + e_k$, where e_k is the unit vector with a 1 in position $1 \leq k \leq r$. Only if this condition holds, we finally apply ancestor-based filtering.

Note that our simple pretest can be conducted much faster than a full-fledged filtering call: it runs in time linear in the size of the given assignments (which is in $O(n)$) whereas ancestor-based filtering wrt each ancestor requires time $O(m^{2.5} + mn)$ for a CSP with n variables and m values.

4.2 Deterministic Lower Bounds

Assume that a call to the ancestor-based filtering procedure reveals that we are at least p edges short of finding a perfect matching in the value dominance graph that was set-up for assignments α and β.[3] Clearly, as was already noted in [15], this means that at least another $p - 1$ variable assignments need to be added to β before filtering can become effective. By adding this information to no-good α, and by keeping track of the depth of the current search node β, we can avoid many useless filtering calls. What is interesting is that we cannot only propagate this information when diving deeper into the tree, but also upon backtracking.

Consider the following example: For no-good α, the check against the current search node in depth d results in a maximum matching with 4 edges missing to be perfect. Then, at depth $d + 4 - 1 = d + 3$, we call for ancestor filtering wrt α again and find that there are still 4 edges missing. Clearly, this means that none of the last 3 branching decisions has brought us any closer to a successful dominance relation with α, and this information can be used even when backtracking up from the current position as illustrated in Figure 1 (A): At depth $d + 2$, for example, we know, even without conducting the filtering call, that the maximum matching must have 4 edges missing. Which implies that, when diving deeper into the tree from depth $d + 2$, ancestor-based filtering cannot be effective until we reach depth $d + 2 + 4 - 1 = d + 5$.

More generally, if at depth $d + p - 1$ we find a maximum matching with $q \leq p$ edges missing to perfection, when backtracking up to depth $d + r - 1$ for some $r < p$, we are sure that there are at least $\max\{r+q, p\} - 1$ more variable assignments necessary before filtering wrt ancestor α can be effective. Consequently, we will not call for ancestor-based filtering wrt α until we reach depth $\max\{d+r+q-1, d+p-1\}$ in the search tree (see Figure 1 (B)).

Note that the above procedure also works if we never get to perform the full filtering procedure at depth $d + p - 1$ because our pretest fails: If the first condition fails, instead of using the number of missing perfect matching edges, we can simply count the number of variable assignments still needed before at least as many variables are assigned in the current search node as are in α. And in the second case, we can count how many more assignments are necessary before each value class signature in α can have become lower or equal to the signatures in the current partial assignment.

[3] The method by which SBDD unifies no-goods ensures that $p > 1$ is only possible for no-goods generated in earlier runs.

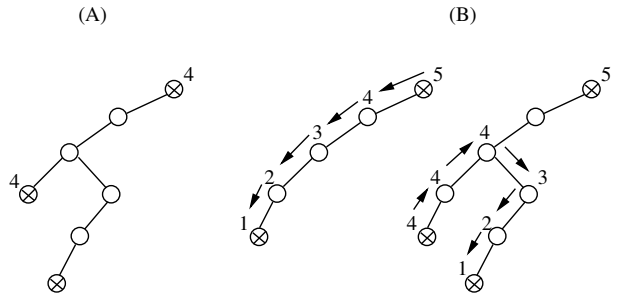


Figure 1: The figures show part of the search tree. The root node is considered to have depth d. At this node, we find that at least 4 (A) or 5 (B) more assignments are necessary before the respective ancestor could dominate the current partial assignment. We dive deeper into the tree without conducting filtering (hollow nodes) wrt the given ancestor until we reach depth $d + 3$ (A) or $d + 4$ (B) where filtering could be effective. In both cases, the call to the filtering algorithm reveals that at least 4 more variable assignments are necessary before dominance can occur. Consequently, the same holds for all ancestors of the respective nodes. By propagating this information up in the tree (see B), filtering is delayed further when branching off from intermediary nodes.

5 Incremental Sibling-based Filtering

Sibling-based filtering requires that we compute the sets of mutually symmetric values that have the same signature under the current partial assignment. Rather than recomputing the signatures of all values and regrouping the values after a branching step has added another variable assignment, we use an incremental data structure for this purpose so as to conduct this type of symmetry related inference as efficiently as possible.

First, let us describe the idea of sparse signatures that are needed to guarantee the worst-case complexity as given in [15]: Instead of writing down entire signatures, for each value we maintain a sparse list that only contains the non-zero entries of a signature, together with the information to which variable partition an entry in the sparse list belongs. To set up this sparse representation from a new partial assignment, we first order the variable instantiations in a given partial assignment according to the partition that the corresponding variable belongs to. This can be done in time linear in the number of variable partitions. In this order, we then scan through the partial assignment and set up the sparse signatures simply by adding one to the last entry if the current variable belongs to the same partition as the last, and by introducing a new non-zero entry if the variable belongs to a new partition.

For the current search node, we group values in the same value partition and with the same signature in the following

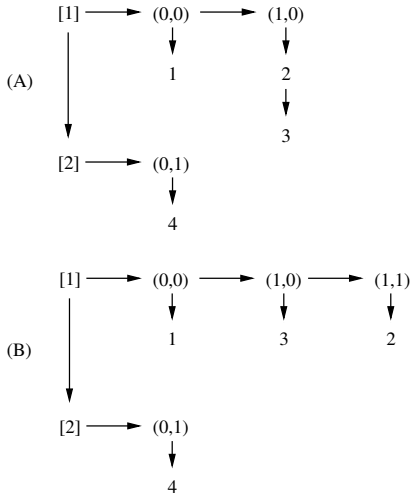


Figure 2: An efficient data structure supporting sibling-based filtering incrementally. The leftmost column depicts value partitions $[1] = \{1, 2, 3\}$ and $[2] = \{4\}$. For both partitions, horizontally it follows a sorted list of signatures that are each associated with all the values underneath them in the current partial assignment. Even though signatures are actually stored in sparse format, we show them explicitly to improve the readability.

data structure. It consists of an array of lists, one for each value partition. Each such list contains, in lexicographic order, the different signatures within the respective value partition. Associated with each signature is yet another list of values in the partition that have the signature, whereby each value holds a pointer to the signature. Note that this data structure allows us to perform sibling based-filtering extremely efficiently. Given a variable, the different values that we need to consider when branching are only the first values in each list of each signature in each value partition.

Now, when branching by assigning a value to some variable, we update the data structure incrementally. Note that the value assigned is the first in its list of values with the same signature. Moreover, the value holds a pointer to its signature. Therefore, we can compute its new signature incrementally, and since the signatures within the value partition are ordered lexicographically, we can also find out quickly to which signature the value needs to be added, whereby we create a new list of values if the value's new signature is not yet in our list. Finally, we remove the value from the list of values for its old signature and add it to the list of values for its new signature, while updating the value's pointer to its own signature.

We illustrate the data structure in Figure 2 on the following example. Assume we are given four variables $X_1, \ldots, X_4$, whereby the first two and the last two are symmetric. Assume further that the variables can take four values $1, \ldots, 4$, whereby the first three are symmetric. Figure 2 (A) shows our incremental data structure for the partial assignment $\{(X_1, 2), (X_2, 3), (X_3, 4)\}$. We see that we can easily pick non-symmetric values simply by choosing the first representative for each signature. In our example, those are the values $1, 2$, and 4. Figure 2 (B) shows the data structure after another variable has been instantiated by adding $(X_4, 2)$ to our assignment. We see that the data structure can easily be adapted by updating the signature of value 2.

	UNBIASED		BIASED			
	15		15		30	
VPC	AllDiff	GCC	AllDiff	GCC	AllDiff	GCC
2	100	100	100	100	100	98-100
3	100	100	100	100	100	52-100
4	100	98-100	100	92	84-96	14-80
5	100	100	88	66	52-82	0-54
6	100	98-100	68	18-32	26-76	0-50
7	94	96-100	26	4-18	6-40	0-50
8	90	88-94	18	0-6	0-16	0-34
9	84	92	0	0-2	0	0-32
10	48	58	0	0	0	0-22
11	16	14	0	0	0	0-22
12	4	0	0	0	0	0

Table 1: Percentages of feasible solutions in the different benchmark sets for different numbers of values per constraint (VPC). We give ranges where even the best algorithm hit the time limit of 600 seconds.

6 Experimental Results

With all the practical enhancements that we introduced in the previous section, we now wish to test structural symmetry breaking in practice. To this end, we need an appropriate benchmark set. Surprisingly enough, despite the large body of work on symmetry breaking, so far we are still lacking a benchmark set which allows us to experiment with symmetry breaking techniques over different regions of constrainedness. Standard benchmarks in the literature are graceful graphs, n-queens, balanced incomplete block designs, or the infamous social golfers, none of which give us a reasonable insight regarding the constrainedness of problem instances. Consequently, until today we lack a comparison of different symmetry breaking techniques over the entire region of constrainedness.

6.1 The Benchmark

We introduce a very simple benchmark generator that produces surprisingly hard instances of piecewise symmetric CSPs: Given a number of variables n and values m, as well as the number of variables n_c and the number of values m_c per constraint, we generate a given number of global cardinality constraints (GCC) over a set of n_c randomly chosen variables and m_c randomly chosen constraints, whereby we enforce that all variables in the constraint together take each chosen value exactly once. We vary the basic concept in the following ways:

- We add one more GCC over all variables and values that enforces that each value be chosen at most 2 times. Alternatively, we add an AllDifferent constraint over all variables and values.
- We draw variables and values uniformly or with a bias such that components with higher indices are more likely to be chosen than those with lower indices.

Tables 1 summarize the properties of our benchmark sets. We consider problems with 15 variables and values and 30

Algo	FO	SO	NO	FR	SR	NR
Sym-Level	Full	Sibl.	None	Full	Sibl.	None
Restarted	no	no	no	yes	yes	yes

Table 2: Overview of the different algorithm variants: Full refers to the variant where we call for ancestor and sibling-based filtering at every search node. Sibling refers to breaking value symmetry only by performing just sibling-based filtering. As a convention, when pretests or delayed filtering are switched off, we add '-P' or '-D' to the name of the algorithm.

variables and values. The number of variables per constraint is fixed at 12 while the number of values per constraint runs from 2 to 12, thus giving us a range of differently constrained instances. Table 1 shows the percentage of feasible instances out of 50 randomly generated ones. In addition, we vary the constraint over all variables and values (GCC or AllDifferent), and we select variables either uniformly or in a biased fashion, while values are always selected uniformly.

6.2 The Contestants

We implemented structural symmetry breaking as explained in the previous sections, whereby we can choose to

- run full SSB filtering for each search node,
- run sibling-based filtering only, or
- perform no symmetry breaking at all.

Additionally, when running full SSB, we can switch both the use of lower bounds and the simple pretest for delayed ancestor-based filtering on or off. The branching variable is determined dynamically by a min-domain heuristic. We can also choose to run the solver with a restart heuristic whereby we choose a linear increase in the fail limits starting with as little as 100. In case of the restarted method, the branching variable is chosen according to a min-domain heuristic over a random subset of 20% of the variables. Table 2 summarizes the settings and names the different contestants that we let compete against one another.

All experiments in this paper were conducted on a 2 GHz AMD Athlon 64 Processor 3000+ CPU with 512 MByte main memory running Linux 2.6.10. Our code was written in C++ and complied by the Gnu g++ compiler version 3.3.5 with optimization flag -O6. As our constraint solver, we used Ilog Concert 2.2 on top of Ilog Solver 6.2.

6.3 The Influence of Pretests and Delayed Filtering

First, we evaluated the effects of the practical improvements that we introduced in this paper. Figure 3 shows the impact of pretesting and delayed filtering on two benchmark sets with very different characteristics. We interrupted method FR after 600 CPU seconds and compare its runtime against FR-PD when the latter executes the same search tree. While all instances in the 15 variable benchmark were solved to completion within the time limit, only about 40% could be solved in the 30 variable benchmark which explains the ruggedness of the curve.

As was to be expected, in both cases we see a beneficiary impact of our techniques to reduce the filtering efforts incurred by SSB: On the 15 variable benchmark, we see that

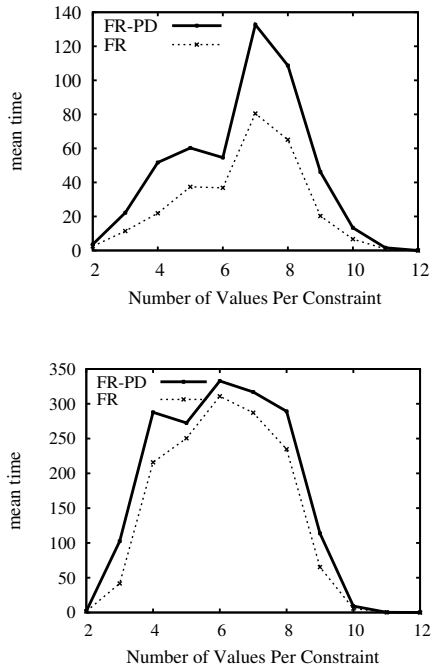


Figure 3: The impact of pretesting and delayed filtering on 50 instances with 15 variables and values, biased variable selection, and global GCC (top) and 30 variables and values, biased variable selection, and global AllDiff (bottom). We compare runtimes (on identical instances) by algorithms FR and FR-PD.

both pretest and delayed filtering save us up to 9000 full symmetry checks, which results in a speed-up of 3 for the overall method. On the 30 variable benchmark, we see an even greater impact: For 4 values per constraint we save more than 175,000 full symmetry checks which results in a speedup of more than 10. As we will see in Section 6.5, these improvements are key to making use of symmetry no-goods in restarted search methods.

6.4 The Impact of Symmetry Breaking

The main objective of this investigation is to study the practicability of SSB. Clearly, symmetry breaking only ever pays off when the work that we have to put in to detect symmetry does not exceed the work that we save in this way. Figure 4 shows three algorithms running three different levels of symmetry breaking in a one-shot deterministic run on instances with 15 variables and values and biased variable selection. Experiments were interrupted after 600 CPU seconds.

We see clearly that SSB (FO) leads to remarkable improvements over a standard CP approach that is unaware of symmetry alltogether (NO). This holds particularly in the critically and over-constrained regions, whereby ignoring symmetry breaking in part or alltogether can be the better option in the very under-constrained region, as we see in the experiment with uniform value selection and global GCC.

Especially the benchmark with biased variable selection and global AllDifferent constraint shows that SSB, despite it's huge computational costs, is very worthwhile and can lead to speed-ups of orders of magnitude. To investigate the effect of

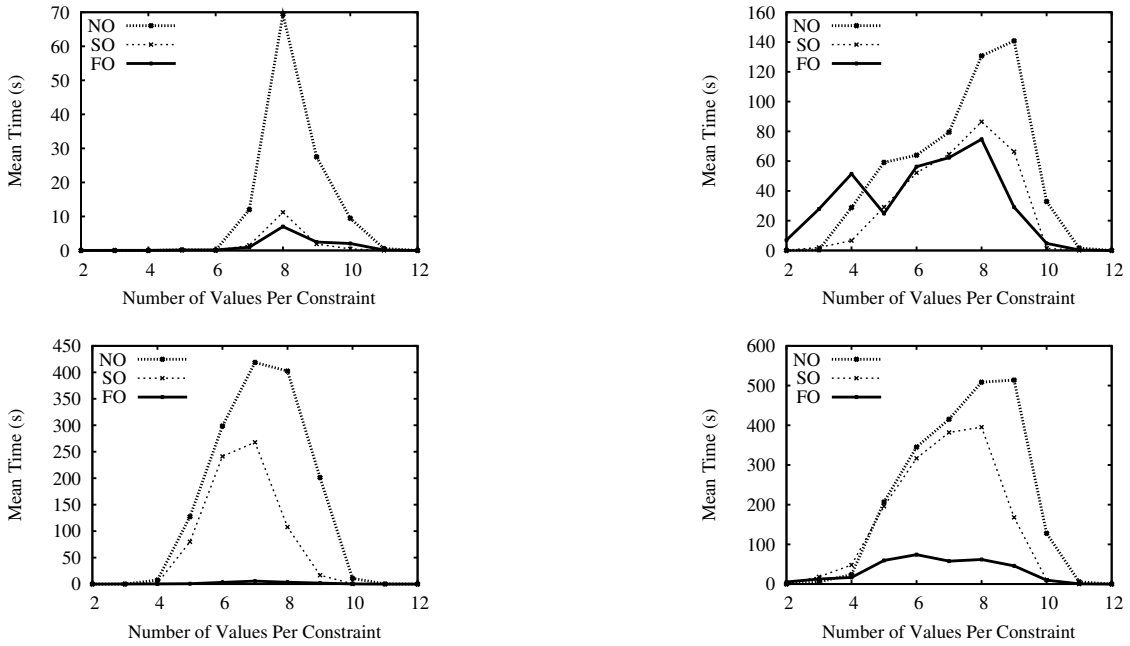


Figure 4: The figures give mean times in seconds of non-restarted algorithms on 50 instances with 15 variables and values, 12 variables per constraint, unbiased (top) or biased (bottom) variable selection, and AllDifferent (left) or GCC (right) as constraint over all variables and values.

symmetry breaking further, in Table 3 we show the number of fails and the time spent per choice point for FO, SO, and NO on two very different benchmark sets with 15 variables and values and 12 uniformly chosen variables per constraint, as well as 30 variables and values and 12 non-uniformly chosen variables per constraint (both have the AllDifferent as global constraint). The table reveils the dramatic change in the characteristic of our algorithm that is introduced by symmetry breaking: NO and SO investigate hundredthousands and millions of search nodes while spending a miniscule amount of time per search node. FO on the other hand spends far more time in every choice point but therefore greatly reduces the number of nodes visited.

6.5 The Impact of Restarts

After seeing that symmetry breaking is beneficial even at the tremendous costs that SSB filtering incurs, we are curious to see how restarts affect the landscape. We show the performance of the restarted algorithms in Figure 5 on the benchmark sets with 15 variables and in Table 4 on the benchmark sets with 30 variables.

The comparison of Figure 5 with Figure 4 shows that the algorithm that is unaware of symmetries can benefit greatl from restarts. In the biased/AllDifferent case, for example, NR is more than 10 times faster than NO. The erratic curves for method NR depict an increase in the variance of the running time. Note that this variance is actually not introduced by the restarts but inherent to a method that can get very unlucky by getting stuck exploring symmetric parts of the search space over and over and over again. The fact that this was not visible in Figure 4 is due to the time limit (that we needed to impose to conduct our experiments within reasonable time) which artificially decreases the variance of the slow algorithm NO. Table 4 also shows very clearly that NR performs far better than NO.

We get a similar picture when comparing the performance of SO and SR. In the set with 15 variables and values, 12 non-uniformly chosen variables per constraint, and one global AllDifferent constraint, for example, SR is about 25 times faster than SO. What is interesting to note is that restarts can actually be counter-productive in the over-constrained region. We suspect that this has to do with the linear increase of fail-limits that we chose for our experiments. Clearly, the shortest proof-tree showing that no solution exists can be quite large for a method that does not eliminate symmetry, and so it naturally takes a lot of restarts to get to that point.

When we perform full SSB filtering, we see that restarts do not help on the benchmark with 15 variables and values. Only as we tackle large and very hard problems, FR starts to outperform FO as can be seen in Table 4 when we consider instances with a global GCC.

This leads to a suprising conclusion: Just breaking value symmetry in combination with restarts is in many cases competitive with full SSB! Compare FR and SR on global AllDifferent instances in Figure 5, or on global GCC instances in Table 4, for example. Of course, SSB is still the clear winner in the critical region on our large benchmark set with 30 variables, but the good performance of restarted sibling-filtering is still astonishing. It is reasonable to conjecture that restarts can actually help reducing the adversary effects of variable symmetry, thus making light-weight value filtering a serious competitor. We believe that this could explain the common assessment that symmetry breaking does not pay off in local search methods [12]: Local search methods naturally explore

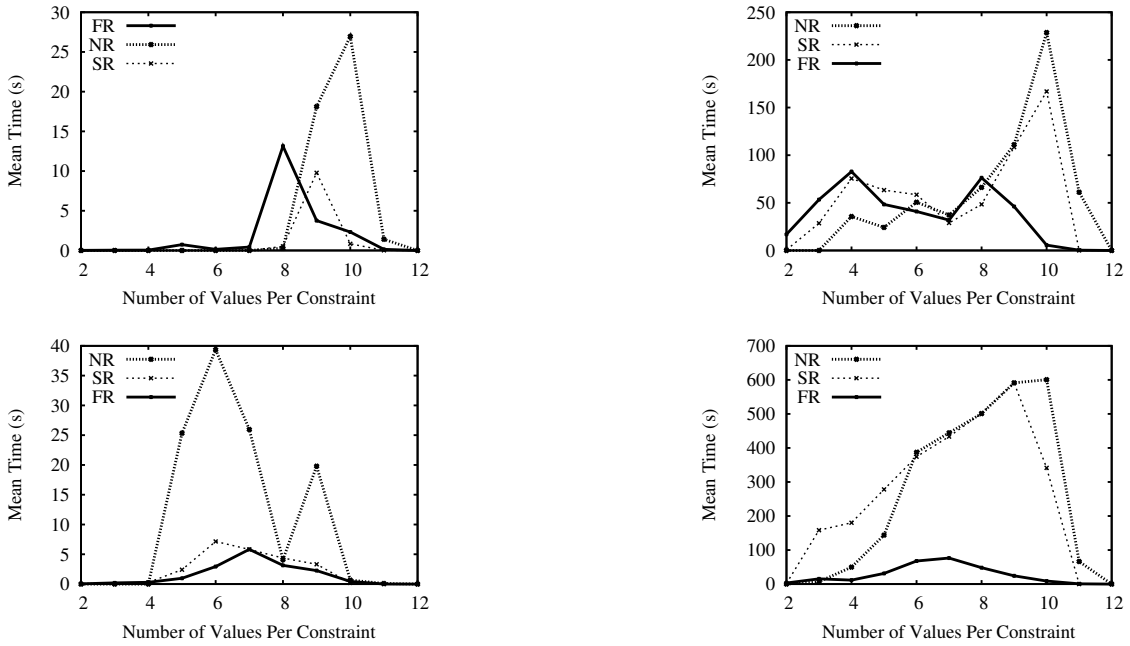


Figure 5: The figures give mean times in seconds of restarted algorithms on 50 instances with 15 variables and values, 12 variables per constraint, unbiased (top) or biased (bottom) variable selection, and AllDifferent (left) or GCC (right) as constraint over all variables and values.

wider parts of the search region. Moreover, the choice how solutions are represented often removes value symmetry directly. When we also take into account that local search is usually applied on underconstrained problems only, then the assessment that symmetry breaking is not worthwhile in local search is not so surprising anymore.

7 Conclusions

We provided practical enhancements of structural symmetry breaking and tested them on the first randomized benchmark set for symmetry breaking experiments. We showed that our practical enhancements lead to substantial speedups that are crucial to make restarts for SSB worthwhile. However, even with these enhancements, restarted methods that break only value symmetry are often almost as competitive as full symmetry breaking.

References

[1] N. Barnier and P. Brisset. Solving the Kirkman's schoolgirl problem in a few seconds. *Proceedings of CP'02*, 477–491, 2002.

[2] C. Brown, L. Finkelstein, P. Purdom Jr. Backtrack searching in the presence of symmetry. *Proceedings of AAECC-6*, 99–110, 1988.

[3] D.A. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, B.M. Smith. Symmetry Definitions for Constraint Satisfaction Problems. *Constraints*, 11(2–3): 115–137, 2006.

[4] J. Crawford, M. Ginsberg, E. Luks, A. Roy. Symmetry-breaking predicates for search problems. *Proceedings of KR'96*, 149–159, 1996.

[5] T. Fahle, S. Schamberger, M. Sellmann. Symmetry Breaking. *Proceedings of CP'01*, 93–107, 2001.

[6] P. Flener, A. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, T. Walsh. Breaking row and column symmetries in matrix models. *Proceedings of CP'02*, 462–476, 2002.

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

[8] I. Gent, W. Harvey, T. Kelsey, S. Linton. Generic SBDD using computational group theory. *Proceedings of CP'03*, 333–347, 2003.

[9] I. Gent and B. Smith. Symmetry breaking in constraint programming. *Proceedings of ECAI'00*, 599–603, 2000.

[10] C.P. Gomes, B. Selman, N. Crato. Heavy-Tailed Distributions in Combinatorial Search. *Proceedings of CP'97*, 121–135, 1997.

[11] Dynamic Restart Policies. H. Kautz, E. Horvitz, Y. Ruan, C. Gomes, B. Selman. *Proceedings of AAAI'02*, 674–682, 2002.

[12] S. Prestwich and A. Roli. Symmetry Breaking and Local Search Spaces. *Proceedings of CPAIOR'05*, 273–287, 2005.

[13] J.-F. Puget. Symmetry breaking revisited. *Proceedings of CP'02*, 446–461, 2002.

[14] C. Roney-Dougal, I. Gent, T. Kelsey, S. Linton. Tractable symmetry breaking using restricted search trees. *Proceedings of ECAI'04*, 211–215, 2004.

[15] M. Sellmann and P. Van Hentenryck. Structural Symmetry Breaking. *Proceedings of IJCAI'05*, 298–303, 2005.

[16] P. Van Hentenryck, P. Flener, J. Pearson, M. Agren. Tractable symmetry breaking for CSPs with interchangeable values. *Proceedings of IJCAI'03*, 277–282, 2003.

	15, unbiased, AllDifferent			30, biased, AllDifferent		
	FO	SO	NO	FO	SO	NO
2	1.1K - 14	120 - 15	67 - 15	17K - 151	42 - 547	17 - 723K
3	2.4K - 17	87 - 23	113 - 16	11K - 1.7K	46 - 17K	19 - 3.7M
4	2.4K - 19	83 - 41	53 - 19	35K - 4.9K	51 - 435K	21 - 6.9M
5	3.6K - 38	39 - 716	24 - 892	37K - 7.9K	57 - 4.3M	26 - 14M
6	5.1K - 32	43 - 368	30 - 2708	32K - 11K	65 - 7.6M	31 - 19M
7	6.6K - 140	44 - 35K	30 - 401K	29K - 11K	74 - 6.5M	38 - 15M
8	9.4K - 720	48 - 233K	33 - 2.08M	28K - 6.9K	84 - 2.6M	49 - 13M
9	7.6K - 329	52 - 37K	36 - 766K	25K - 2.9K	104 - 215K	63 - 9.6M
10	6.4K - 319	61 - 7600	40 - 239K	16K - 402	131 - 5.6K	78 - 5.7M
11	3.4K - 43	76 - 132	46 - 12.5K	4.5K - 42	112 - 90	90 - 349K
12	X - 0	X - 0	X - 0	X - 0	X - 0	X - 0

Table 3: Times per choice point in micro seconds and number of search nodes (averages over 50 instances per data point).

	AllDifferent						GCC					
	One-Shot			Restarted			One-Shot			Restarted		
	FO	SO	NO	FR	SR	NR	FO	SO	NO	FR	SR	NR
2	100	100	98	100	100	100	46	62	86	78	100	100
3	100	100	90	100	100	100	36	36	78	46	100	100
4	86	98	76	76	92	100	41	40	55	41	88	47
5	60	72	40	56	56	88	30	30	24	28	54	24
6	52	23	4	54	18	51	14	10	2	12	8	10
7	61	41	2	65	20	9	14	4	2	10	12	6
8	80	84	0	82	52	0	4	0	0	14	0	0
9	96	100	0	98	76	0	24	2	0	34	0	0
10	100	100	36	100	100	0	26	10	0	76	2	82
11	100	100	96	100	100	100	52	50	48	70	100	100
12	100	100	100	100	100	100	100	100	100	100	100	100

Table 4: Histogramm for the benchmark sets with 30 variables and values, 12 variables per constraint, and biased variable selection. The first column gives the number of values per constraint, the numbers in the table the percentage of instances solved within 600 seconds (50 instances per data point).

Speeding up Weak Symmetry Exploitation for Separable Objectives

Roland Martin
Darmstadt University of Technology
Algorithmics Group
64289 Darmstadt, Germany,
martin@algo.informatik.tu-darmstadt.de

Abstract

We consider an algorithm that enables us to reduce the number of solutions to consider for weakly symmetric problems. The idea is to store partial permutations during the solving process and re-use them for future solutions thereby reducing the number of weakly symmetric solutions to consider for these solutions. The idea can be applied to problems where the weak symmetry is introduced by a separable objective function. We present the theoretical soundness of the idea and a prototype algorithm that could be in-cooperated as a global constraint to the constraint solver.

1 Introduction

Weak symmetries act only on a subset of the variables and the weakly symmetric solutions satisfy only a subset of the constraints of the problem. Therefore, weak symmetric solutions preserve the state of feasibility only with respect to the subset of variables the weak symmetry acts on and only for the constraints these variables satisfy. If two solutions s_1 and s_2 are weakly symmetric there is in general no symmetry function that maps s_1 to s_2 or vice versa. The solutions s_1 and s_2 are only symmetric with respect to the variables the weak symmetry acts on. Using a modelling approach introduced in [1] we are able to break the weak symmetry without losing solutions. Nonetheless all weak symmetric solutions have to be considered for the problem. That means that the whole equivalence class for each weak symmetric solution has to be considered in order not to lose solutions. Weak symmetries occur in many fields of applications and have already been discovered and identified in planning, scheduling and model checking (see [2] - [6]). In particular real world optimisation problems contain weak symmetries. Often the objective function makes a symmetry weak.

In many cases also the objective function is separable in the columns/rows of a variable matrix. That means that for each column/row a partial objective value can be computed that is independent from all other variable instantiations of the matrix. The objective value then is an aggregation (for example the sum) of these partial values.

If a separable objective function induces a weak symmetry on a problem then our approach can be used to save time in investigating the equivalence class for weak symmetric solutions. The idea is to store partial results from investigating the equivalence class of certain solutions and re-use them for the investigation of so-called neighbouring solutions. Thereby reducing the number of solutions to check explicitly.

Section 2 introduces the concepts of weak symmetry and weak symmetry breaking while Section 3 introduces in separable objectives. In Section 4 the approach for exploiting weak symmetries and separable objectives is explained and theoretical soundness is given. Section 5 shows an example where the approach could be applied and Section 6 reasons about the usability and limitations of the approach. The paper concludes and gives an outlook to future work in Section 7.

2 Weak Symmetry

Weak symmetries are rather new and not widespread in the constraint programming scene by now. To make the paper self-contained we take over the key definitions and facts of weak symmetries from [1]. For further investigation and a running example see also [1].

2.1 Weak Symmetry Definition

Weak symmetries act on problems with special properties. To characterize weak symmetries we first define weakly decomposable problems. In a weak decomposition of a problem all variables and constraints that are respected by the weak symmetry are gathered in one subproblem.

Definition 1 (Weakly Decomposable Problem) *A problem $P = (X, D, C)$ is* **weakly decomposable** *if it decomposes into two subproblems $P_1 = (X_1, D_1, C_1)$ and $P_2 = (X_2, D_2, C_2)$ with the following properties:*

$$X_1 \cap X_2 \neq \emptyset \quad (1)$$
$$X_1 \cup X_2 = X \quad (2)$$
$$C_1 \cup C_2 = C \quad (3)$$
$$C_1 \cap C_2 = \emptyset \quad (4)$$
$$C_2 \neq \emptyset \quad (5)$$
$$D_1 = pr_1(D) \quad (6)$$
$$D_2 = pr_2(D) \quad (7)$$

where pr_i denotes the projection to the subspace defined by the subset X_i of the variables in P.

The first condition states that P_1 and P_2 contain a subset of shared variables (namely $X_1 \cap X_2$). These variables have to assume the same values in both subproblems to deliver a feasible solution to P. Therefore they link both problems. Without that restriction the problem would be properly decomposable. The second and third condition states that none of the variables and constraints of the original problem P are lost. Furthermore the third and fourth condition state that C_1 and C_2 is a partition of C. Basically this is not necessary for feasibility. A constraint could be in both subsets (if defined on $X_1 \cap X_2$ only) but would be redundant for one of the problems because the solution to the other subproblem would already satisfy this constraint. Therefore, this is just a question of efficiency. The fifth condition states that P_2 is not allowed to be unconstrained. However, note that this restriction does not hold for P_1. This is since we want to group the symmetric data in P_1 and a problem without constraints is perfectly symmetric. Every CSP is weakly decomposable, and usually there will be multiple weak decompositions. However, we concentrate on weak decompositions where the weak symmetry acts as a proper symmetry on P_1.

Definition 2 (Weak Symmetry)
Consider a weakly decomposable problem P with a decomposition (P_1, P_2).

A symmetry $f : X_1 \rightarrow X_1$ on P_1 is called a **weak symmetry on** *P with respect to the decomposition (P_1,P_2) if it cannot be extended from X_1 to a symmetry on X.*

The intention of the decomposition of the problem is that X_1 contains all symmetric variables (and only those) and X_2 contains the rest of the variables. The gain is that we get a subproblem where the weak symmetry affects all variables and all constraints (P_1) and therefore acts as a proper symmetry on it and one subproblem that is not affected by the weak symmetry (P_2).

2.2 Breaking Weak Symmetries

The challenge in weak symmetry breaking is actually not the symmetry breaking part but not to lose solutions by breaking the weak symmetry. As mentioned earlier the weak symmetry is a proper symmetry on P_1, and any method of symmetry breaking can be used. However, by breaking the symmetry on P_1, any solution s_{P_1} of P_1 will represent its equivalence class of solutions. All these solutions have to be considered when determining a solution to P. Even if s_{P_1} does not satisfy P_2 a different solution $\pi(s_{P_1})$ in the same equivalence class may satisfy P_2, where $\pi(s_{P_1})$ is a permutation of s_{P_1}.

Therefore we need a way to represent all these solutions in the search process. We introduce a new variable that identifies which element of the equivalence class is represented in the further search process. This variable is the SymVar.

Definition 3 (Symmetry Variable)
Consider a CSP $P = (X, D, C)$ with a weak decomposition (P_1, P_2) and a weak symmetry f on P.

A **symmetry variable (SymVar)** *$\pi \in S[X_1]$ represents the group of symmetric solutions of f in P_1. Its domain is the symmetric group on X_1, denoted by $S[X_1]$.*

If the SymVar is the identity then the solution passed to P_2 is the one found in P_1. In any other case the permuted solution of P_1 (which is equivalent with respect to the weak symmetry) is passed to P_2. The solution of P_1 together with the assignment of the SymVar represents a partial variable assignment to P_2 and P. It is checked whether it also satisfies the constraints of P_2 and if so all variables in $X_2 \backslash X_1$ are assigned for finding a solution to P_2. If the partial assignment does not satisfy P_2 a different element of the equivalence class is considered by a different value for the SymVar. If none of the elements satisfy P_2 a new solution to P_1 is sought. This way the whole problem is investigated and no solution is lost. Note that only for solutions of P_1 the SymVar is instantiated.

Theorem 1 (Solution Preservation)
The solution space of P is totally reflected by the decomposition (P_1, P_2) and a SymVar π such that every solution of P can be uniquely represented by a solution to P_1, an assignment to the SymVar and a solution to P_2.

See [1] for a proof.

In practice this concept of a single SymVar as a representative is not supported in constraint programming solvers on the level of modelling. Therefore instead of one variable we use a set of variables. A feasible variable assignment to these variables then represents a specific element of the equivalence class.

Definition 4 (P_{sym})
Consider a CSP $P = (X, D, C)$ with a weak decomposition (P_1, P_2) and a weak symmetry f on P.
$P_{sym} = (X_{sym}, D_{sym}, C_{sym})$ is a subproblem of P that models the weak symmetry f. X_{sym} is the set of SymVars representing the variables of P_1. D_{sym} is the domain for all SymVars and C_{sym} is the set of constraints that model the symmetric group induced by f. A solution of P_{sym} represents an element of the symmetric group induced by f.

If the weak symmetry is a permutation of n elements there are n SymVars with a domain of $\{1, \ldots, n\}$ and an alldifferent constraint ensuring that every feasible assignment to X_{sym} is a permutation.

To solve P we consider the partial solution $s_{P_{sym}}$. When a solution is found, the search backtracks and reconsiders values for the SymVars to determine a new solution. All these solutions are symmetric equivalents to the solution s_{P_1}. Only when the search backtracks and variables in X_1 are reconsidered, a solution for a different equivalence class can be found.

By using SymVars we can break the symmetry in P_1 but do not lose any symmetric solution in an equivalence class.

3 Separable Objectives

3.1 Prerequisites

For the rest of the paper we consider the following problem structure: The CSP $P = (X, D, C)$ is an optimisation problem, i.e. there exists an objective function f that assigns each solution S of P an objective value $f(S) = v$ which leads to a

ranking of all solutions. P weakly decomposes to P_1 and P_2 whereby the weak symmetry in P_1 is a column permutation of a search variable matrix $\chi^{m\times n} \subset X$. P_2 just consists of the objective function as a constraint on χ: $P_2 = (\chi, D_\chi, f)$. A solution S to P consists of the permutation s_π of the solution s to P_1 and the objective value v associated to s_π via the function f. The column permutation symmetry in P_1 is broken by a symmetry breaking method. P_{sym} consists of investigating the symmetric equivalents using n SymVars sv_i representing the columns of χ. An assignment to all SymVars sv_i therefore models s_π which is a solution to P_{sym}. For our purpose we do not care which symmetry breaking method is chosen in P_1. Symmetry and Weak Symmetry Breaking do not conflict with each other [1] so we can choose the most effective method for our purpose. In [1] we used a modelling approach to break the symmetry. Based on this method we describe our approach.

Convention: Small capital variables s are solutions to the subproblem P_1 while large capital variables S are solutions to the whole problem P.

We also define a *partial permutation* in the following. For our purpose a partial permutation is a permutation of just some consecutive variables while the rest of the variables is not assigned yet. This represents a search state for P_{sym} where some SymVars are assigned already and others are still unassigned.

Definition 5 (Partial Permutation)
Consider a permutation π of n variables. A partial permutation π_i is an assignment of the fir sti variables on the domain $1, \ldots, n$.

A partial permutation π_i implies that i values have been assigned to the first i variables (in our case the SymVars) and $n - i$ values have not been assigned.

3.2 Separable Objectives

Separable objectives have the special feature that the objective function f itself can be broken down to independent subfunctions $f_1 \ldots, f_n$, each defined over a variable set $\chi_i \subset \chi$. The variable sets of the subfunction form a partition of χ.

For our purpose we regard only functions where the subsets χ_i form a structure like the columns or the rows of χ.

Definition 6 (Separable Objective)
An objective function stated over a search variable matrix χ is separable in the columns (rows) if

- *the contribution of an assigned column (row) to the objective value is independent from the assignments of all other columns (rows)*
- *the objective value can be computed from the separate contributions of all columns (rows)*

Note that especially in real-world optimisation the are a lot of problems that introduce weak symmetries are are separable in the desired way since the optimisation function itself introduces the weak symmetry.

3.3 Separable Objectives and Weak Symmetries

If the variable matrix χ comprises for example a column permutation symmetry on P_1 but not on P the symmetry is weak on P. For the sake of simplicity we consider the weak symmetry to be a column permutation from now on. If the objective function is also separable in the columns we can store the partial permutations and the achieved partial objective value. These partial permutations of a solution s can be re-used for a solution s' if s and s' are *neighbouring* solutions.

Definition 7 (neighbouring Solutions)
Given two solutions s and s' to a problem P. Each solution consists of a search variable matrix $\chi^{m\times n}$.

s and s' are neighbouring solutions *if the following holds:*
$\exists i \in \{1, \ldots, n\} \forall j \in \{1, \ldots i\} : s_j = s'_j$*, where s_j is the j-th column of the solution s and analogously for s'.*

Definition 8 (Neighbourhood Degree)
Given two neighbouring solutions s and s' to a problem P. Each solution consists of a search variable matrix $\chi^{m\times n}$.

The highest index i for which s and s' are neighbouring is called the neighbourhood degree *of s and s':*
$nbhDeg(s, s') = \max_{i\in\{1,\ldots,n\}} \forall j \in \{1, \ldots i\} : s_j = s'_j$

Note that we define neighbourhood as a successional feature. If two solutions are neighbouring for a certain index i than they are also neighbouring for all $j < i$. The reason for that is to achieve a trade off between the efficiency and the space complexity of the proposed method in this paper. It is without loss of generality possible to define the neighbourhood relation just for single and not for successional indices. But this would result in a super-exponential space consumption such that the method would not be applicable in practice.

In our scenario s and s' are solutions to P_1 and they are subject to column permutation to determine the solution for the CSP P. That means that the whole equivalence class for all solutions to P_1 have to be checked explicitly. In fact for each solution $n!$ permutations have to be considered.

Consider now that s and s' are neighbouring with the degree k. In this case the permutation of the first i columns is part of both solutions s and s'. Without loss of generality s is found before s' in the search. The idea is to re-use the results of the partial permutations of the first k columns from s for the computation of the permutations to s'. By doing so the number of permutations that have to be checked explicitly for s' reduces from $n!$ to $\frac{n!}{k!}$ (See Section 4).

Therefore we store the partial permutations as well as the achieved objective value when checking all permutations π of s. In fact, we do not need to store *all* partial permutations but only *dominating* partial permutations.

Definition 9 (Dominating Permutation)
A permutation π dominates an other permutation π' with respect to s if

$f(s_\pi) \geq f(s_{\pi'})$*, where $f()$ is the objective function.*

If π dominates all other permutations with respect to s, π is a dominating permutation.

Definition 10 (Dominating Partial Permutation) *Consider π_i and π_i' to be partial permutations.*
π_i dominates π_i' with respect to s if

- *s_{π_i} and $s_{\pi_i'}$ have assigned the same set of values (but to different variables)*
- *$f(s_{\pi_i}) \geq f(s_{\pi_i'})$, where $f()$ is the objective function*

If π_i dominates all other partial permutations $\bar{\pi}_i$ that have assigned the same set of values, π_i is a dominating partial permutation with respect to s.

In fact we store for each subset of the set of columns one dominating partial permutation. We can do this because we are only interested in the partial permutation that achieves the best objective value for each subset of the columns. The number of dominating partial permutations is considerably lesser than that of all partial permutations.

Storing the partial permutations can be done during the search process of P_{sym}. After each instantiation of a SymVar it is checked whether this partial permutation dominates a previous found partial permutation on the set of assigned columns. If so the new partial permutation is stored.

If all permutations π have been considered for s, search backtracks to find a new solution s' to P_1. The neighbourhood degree k of s and s' is determined and the permutation of s' is started. Since $nbhDeg(s, s') = k$ the first k columns of s and s' are identic. Due to the separable objective function f that means that a partial permutation of the first k columns achieves the same partial objective value for s and s'. Since we already performed these partial permutations on s we do not want to perform them again but use the stored dominated partial permutations.

Therefore we instantiate only the last $n - k$ SymVars, instead of all n SymVars. When all these SymVars are assigned there are k remaining values that were not assigned. For these values we recall the stored dominating partial permutation which is applied to the remaining k unassigned SymVars. Together this forms a permutation of all columns. When the objective value is determined the search backtracks and performs search on the last $n-k$ SymVars. For each permutation of the last $n - k$ SymVars the dominating partial permutation for the not assigned columns are recalled and the permutation problem is reduced to $\frac{n!}{k!}$ permutations to check.

The gain in the method is not only the reduction for *one* other solution. But it holds for each solution $\bar{s}$ that is neighbouring with s. Therefore the stored partial solutions can be used for each $\bar{s}$. Depending on $nbhDeg(s, s')$ more or less permutations can be omitted for $\bar{s}$.

All solutions with a neighbourhood degree of 0 to any previous solution are called *cardinal solutions*. If $nbhDeg(s, \hat{s}) = 0$ that means that the first the first column in both solutions is different: $s_1 \neq \hat{s}_1$.

Definition 11 (Cardinal Solution) *A solution s that has a neighbourhood degree of 0 with all previous found solutions is called a* cardinal solution.

Only for cardinal solutions partial permutations are stored whereby memory is erased with each new cardinal solution.

4 Approach

As outlined before the idea of the approach is to store and re-use partial information from solutions investigated before. We will describe the approach in this section more detailed. It consists of two phases. The first is storing partial solutions during solving P_{sym} for cardinal solutions. The second is calling these stored data for solving non-cardinal solutions. We introduce a variable ordering on P_1 and P_{sym}. This has to be done in order to find the solutions in the desired order.

4.1 Variable Ordering for the Approach

Ordering on P_1

For our approach we consider a fixed variable ordering for the variables in χ. The matrix is assigned columnwise beginning with the smallest index 1 up to n.

The reason for that is that we want the backtracking in a way such that

- as many columns as possible keep fully instantiated
- if a value in a column is backtracked then in the column with the highest index that still has variables assigned

By doing so we achieve the following:

1. all solutions neighbouring to a cardinal solution s are found consecutively
2. the neighbourhood degree between s and any new solution is decreasing (which means that the highest neighbourhood degree is found first)
3. once a solution $\hat{s}$ is found that is not neighbouring with s (i.e. the neighbourhood degree between s and $\hat{s}$ is 0) no further solution is neighbouring with s.

This way the assignment of the columns change from "right to left" during the search which produces the desired feature of decreasing neighbourhood degree.

Theorem 2 (Neighbourhood Decrease) *Consider the variable ordering in P_1 for χ to be performed columnwise increasingly. Than the solutions are found in a way such that the neighbourhood degree of a cardinal solution s and any solution s' found before the next cardinal solution $\hat{s}$ will never increase.*

Proof. If the variables are assigned in the proposed order than the assignments in the columns will change from right-to-left changing first columns with higher indices. Consider for the cardinal solution s and a solution s' that the neighbourhood degree is k. Any solution s'' found after s' has at least one of the columns with an index lesser than or equal k changed in comparison to s. Therefore the neighbourhood degree cannot increase for the solutions between two cardinal solutions.

As soon as the first column is reconsidered the neighbourhood degree to all previous found solutions is 0. This trivially holds for the first solution as well.

Ordering on P_{sym}
We consider here fixed but different variable orderings depending on the kind of a solution (cardinal or non-cardinal). For cardinal solutions s we assign the SymVars increasingly from sv_1 to sv_n. For a non-cardinal solution s' with $nbhDeg(s, s') = k$ we assign the SymVars decreasingly from sv_n to sv_{k+1}.

Note that the variable ordering for cardinal solutions is no limitation. All permutations of s have to be considered anyway in order not to lose solutions. We save i-elementary subsets of the set of columns with the property that the first i SymVars are instantiated during the search. This is done to save storing capacity. By doing so for each $2 \leq i \leq n-1$ we store all i-elementary subsets of the set of columns. The i-elementary subsets stored represent all partial solutions of a permutation of the first i SymVars on the set of columns.

When P_{sym} is exhaustively investigated for each such a subset a dominating partial permutation is stored.

Theorem 3 (Dominating Permutation)
After all permutations are performed for a cardinal solution s a dominating partial permutation is stored for each subset of the columns.

Proof. For a partial permutation $\pi_c(s)$ the set of assigned columns c is determined. For this subset it is checked whether $\pi_c(s)$ achieves a better objective value than the best found partial permutation $\pi'_c(s)$. If so, $\pi_c(s)$ is stored since it dominates $\pi'_c(s)$. When all permutations are performed for each subset of the columns a partial permutation is stored. Since only dominating permutations are stored and the search is exhaustively the last stored permutation in each subset is dominating.

For non-cardinal solutions we assign only SymVars with a index higher then k and complete the partial solution by calling the relevant stored partial permutation for the first k SymVars.

4.2 Storing Partial Permutation

Partial permutations are just stored for cardinal solutions. This is in particular the first solution s found for P_1. The next cardinal solution $\hat{s}$ is the first found that is not neighbouring with s (i.e. $nbhDeg(s, \hat{s}) = 0$). The next cardinal solution $\hat{\hat{s}}$ is the first found that is not neighbouring with $\hat{s}$ and so on. All solutions found between two cardinal solutions are neighbouring with the first found of these two. For a cardinal solution s all permutations of s have to be considered in order not to lose solutions. For all other solutions only partial permutations have to be considered since the rest of the permutation is taken from the cardinal solution. The number of partial permutations to consider for a solution depends on the neighbourhood degree of this solution and its cardinal solution.

Process of Storing
Consider a cardinal solution – without loss of generality the first found solution – s to P_1. To find a solution to P, s has to be permuted. Therefore the symmetry variables are assigned. The assignment is done such that the columns of the matrix χ, represented by the SymVars, are assigned increasingly. After each instantiation of a SymVar the partial permutation represented by this partial assignment is stored if it dominates all previously found solutions on the set of assigned values. More specifically the objective value and the concrete assignment of the SymVars is stored. Since the objective is separable the objective value can be obtained.

Example: Consider the following partial SymVar assignment: $sv_1 = 3, sv_2 = 1, sv_3 = 4$. This means the first column of χ is permuted to the third column and so on. Consider that the objective value for this partial assignment is 34. Then the data $< (3, 1, 4), 34 >$ is stored for the partial permutation. If the partial assignment is extended by $sv_4 = 6$ achieving a objective value 42, then $< (3, 1, 4, 6), 42 >$ is stored for the partial permutation.

If a different partial permutation achieves a higher objective value than the old one it is overwritten by the better one.

Consider the example above and a new partial SymVar assignment: $sv_1 = 4, sv_2 = 3, sv_3 = 1, sv_4 = 6$ achieving an objective value of 50. The old partial assignment $< (3, 1, 4, 6), 42 >$ is overwritten by $< (4, 3, 1, 6), 50 >$.

Although there are $n!$ permutations the number of dominating partial permutations to store is lesser.

Theorem 4 (Highest neighbourhood degree k)
For a cardinal solution s with n columns it is sufficient to regard only a neighbourhood degree of $2 \leq k \leq n-1$.

Proof. A neighbourhood degree of 1 does not have to be regarded since in this case there is only one search variable left and there exists only one value for this variable due to the permutation. That means that a solver does automatically assign the value and compute the objective value. Therefore there is no use in storing one-elementary subsets. A neighbourhood degree of n is not possible because this would mean that both solutions s and s' have only identic columns which means that $s = s'$. This is impossible since a constraint solver can't find the identic solution again.

That implies that only dominating partial permutations for all 2 to $n-1$-elementary subsets are to be stored. For each subset one dominating partial permutation is stored.

Theorem 5 (Storing Capacity)
The size to store all dominating partial permutations is $2^n - n - 2$.

Proof. For each subset of the columns one dominating permutation has to be stored. There are $\binom{n}{i}$ subsets of the size i. We do not need the subsets of size $0, 1$ and n due to Theorem 4. Therefore there are $\sum_{2 \leq i \leq n-1} \binom{n}{i}$ subsets which equals $2^n - n - 2$.

Since in the applications we are investigating n is bound to be 20 at most the storing capacity is a practical amount and the approach is not only theoretically but can be applied.

4.3 Applying Stored Partial Permutations

The stored partial permutations for a cardinal solution s can be used for each solution s' with $nbhDeg(s, s') > 0$. When s' is found and $nbhDeg(s, s') = k$ the first k columns do not have to be permuted anew.

First a permutation of the last $k + 1, \ldots, n$ SymVars is sought. Then it is determined which values of the columns are not assigned to these SymVars. For these values a dominating partial permutation of the first k SymVars is re-called from the stored data. Since the stored partial permutation for these values is dominating it represents an optimal solution for this subproblem.

Therefore the problem P_{sym} for non-cardinal solutions reduces to investigate only $\frac{n!}{k!}$ permutations instead of $n!$.

Theorem 6 (Reduction for Non-cardinal Solutions) *Given a cardinal solution s and a solution s' with $nbhDeg(s, s') = k$.*

The number of permutations that have to be explicitly investigated for s' reduces from $n!$ to $\frac{n!}{k!}$.

Proof. Due to $nbhDeg(s, s') = k$ the first k columns of both solutions are identic. Therefore only the last $n - k$ columns of s' have to be permuted on the n columns of the matrix. For the remaining k free columns the stored dominating permutation can be taken. Therefore the number of explicitly investigated solutions is $\frac{n!}{k!}$.

4.4 Related Work

We use a domination criterion to reduce the size of solutions to store. The domination criterion can be used since the objective function is separable. There are other approaches that use dominance to speed up the search in different ways. SBDD (Symmetry breaking by dominance detection) [8] for example checks whether a current search state is dominated by a previously found search state. Focacci and Shaw [10] prune search branches that are dominated by other using local search. Smith [9] uses no-good recording to detect whether current search states lead to the same remaining subproblem as previously investigated search states. So far we do not use domination to reduce the number of permutations to consider. But the ideas in [9] could be incooperated to reduce the number of permutations to consider for cardinal solutions. This would mean to alter the search strategy and although a smaller search space is to be investigated for the cardinal solutions it is likely that the first solution is found later which may be a problem in online-optimisation. But it is definitely worth investigating.

5 Example for a Problem with Separable Objectives

We use a problem from the field of automated manufacturing to demonstrate our ideas. The problem is more detailed described in [1; 7]. For our purpose we consider a relaxed subproblem for the sake of simplicity.

5.1 Problem Description

In the problem certain components must be mounted on PC boards by a mounting machine consisting of several mounting devices. The task is to maximise the workload of the whole machine. We concentrate only on a subproblem of the whole solving process. That is to find a setup of component types for the individual mounting devices to maximise the potential workload.[1]

The machine consists of several mounting devices. Each mounting device has access to a set of component types (called setup) that are to be mounted on the PC boards. In addition each mounting machine has only access to a part of the PC board layout and can therefore only mount components inside this visibility area. The PC board layout is specified by a list of mounting tasks. A mounting task is specified by a component type and a position where to mount this component type.

The problem is modelled as follows: The machine is represented by an $m \times n$ variable matrix $\chi^{m \times n}$ where m is the number of different component types that can be assigned to a mounting device and n is the number of mounting devices on the machine. The domain of variables $a_{ij} \in \chi$ is the set of component types. An assignment $x_{ij} = k$ means that a component of type k is placed on the mounting device j in the ith slot.

The constraints:

- No component type may be assigned more than once to a column
- Certain component types may not be assigned together in a column
- Each component type achieves a certain workload when assigned to a column. The workload differs from column to column. This represents the visibility of the mounting device.

In the real-world the matrix χ would have about 10 rows and 6 to 20 columns. Where the most common case is a matrix with 12 columns.

5.2 Weak Symmetry and the Separable Objective

The columns of the matrix χ can be permuted which doesn't change feasibility. But the assigned component types achieve a different potential workload. Therefore the column permutation is a weak symmetry.

The objective function is separable in the columns since the potential workload can be determined for each column separately.

A drawback in the problem is that pruning due to objective value bounding for P_{sym} is not very effective. The reason is that we maximise the objective value and the contributions of each assigned column is strictly positive. That means that mostly the majority of the SymVars have to be instantiated before pruning can be performed.

[1] The actual workload assigned to the devices is a subset of the workload determined in this subproblem. But the higher the possible workload the higher the degree of freedom for the concrete assigning problem not considered here.

5.3 Neighbourhood of the Problem

In this problem solutions have a rather high degree of neighbourhood. This is due to the fact that only few changes in the setup constitutes a new solution. This way the neighbourhood degree decreases slowly such that the time spent for storing solutions for a cardinal solution is outperformed by the saving of performing permutations.

5.4 Efficiency of Applying the Approach

This discussion is held from a theoretical point since the technique has not been applied to the problem yet. Still due to the investigations in [1] we have a lot of knowledge about the structure of the solution space. As mentioned before the neighbourhood degree is very high in the problem. In practice a lot of solutions just differ by two or three columns. In a standard instance with 12 columns that would mean a reduction from 12! to $12^2 - 12 = 132$ permutations for several solutions. The number of solutions is rather high in the problem. This means that even for instances with 12 columns it is not possible to solve the problem exhaustively within reasonable time. Using our method for weak symmetry exploitation a much larger number of solutions can be investigated or the problem could be solved exhaustively for smaller instances which is a large improvement for the problem.

6 Extensions of the Approach

Here we consider some extensions and variations that could be used for the approach. We discuss the advantages and disadvantages of each idea.

6.1 Neighbourhood as a Discrete Feature

We limit ourself to regard neighbourhood as a successional feature. This is done to make the approach applicable. The memory consumption is growing super-exponential otherwise which would allow only very small instances to be solved.

Advantages

If the neighbourhood is defined discrete, i.e. columns do not have to be successional to count for the neighbourhood degree we do not have to impose a variable ordering on P_1 that is that strict. Columns do not have to be considered increasingly but can be assigned arbitrarily as long as all variables of a column are assigned successively.

Disadvantages

The memory consumption is much higher for saving all dominating partial permutations. This is due to the fact that in the successional neighbourhood the values for each k-elementary subset are only assigned to the SymVars $sv_1 \ldots, sv_k$. For a discrete neighbourhood these values could be assigned to any k SymVars. Also there are more saving operations which consume time during the search.

6.2 Imposing a Lower Bound k_{min} for the Neighbourhood Degree

We limit ourself to store data only for cardinal solutions. But since the neighbourhood degree is decreasing during search more and more efficiency is lost. This happens for solutions s, s', s'' with $nbhDeg(s, s'') < nbhDeg(s', s'')$, whereby s is the cardinal solution. In this case the permutation reduction would be better if s' was the cardinal solution. It is possible to impose a lower bound for the neighbourhood degree k_{min} such that the saving for further solutions is higher. That would mean that a solution s' with $nbhDeg(s, s') < k_{min}$ to a cardinal solution s is announced a new cardinal solution and partial permutations for s' have to be stored.

Advantages

Imposing a lower bound on the neighbourhood degree k_{min} would guaranty that for each non-cardinal solution at most $\frac{n!}{k_{min}!}$ permutations have to be performed.

Disadvantages

For the newly announced cardinal solutions storing operations have to be performed which cost time. Fortunately the storing capacity has not to be extended. Moreover, if for a solution s' and a cardinal solution s it holds $nbhDeg(s, s') = k < k_{min}$, the k-elementary subsets do not have to be computed again. Only subsets with more than k elements have to be stored anew. But for s' $n!$ permutations have to be performed. It is not possible to use the stored k-elementary partial solutions and extend them to an optimal permutation. This way solutions could be lost.

6.3 Imposing on Upper Bound k_{max} for the Neighbourhood Degree

On the other hand we do not restrict the maximal degree of neighbourhood k_{max}. Since the memory consumption is exponential in the neighbourhood degree it may be necessary to impose an upper bound. That means that only neighbourhood degrees up to k are respected. Clearly this is a loss of efficiency for the method but it makes it applicable.

Advantages

The clearest advantage is that the extra memory consumption is under control which makes the approach applicable. Although this clearly limits the theoretically achievable efficiency it also offers us a chance of pruning in P_{sym}. Only permutations of the first k_{max} variables are stored. We only have to investigate the assignment of these variables exhaustively since we do not have to keep track of the optimal partial assignments. That gives us the freedom to prune partial permutations beyond an assignment of k_{max} variables.

Disadvantages

Clearly not full efficiency could be achieved since the solutions of the problem may have neighbourhood degrees greater than k_{max} such that theoretically more permutations could be avoided to perform.

7 Conclusions and Outlook

We proposed a new algorithm that exploits weak symmetries for separable objectives in a way such that the number of permutations to perform can be reduced for certain solutions. By spending a manageable amount of memory partial permutations are stored for so called cardinal solutions. These data is used to save performing permutations for non-cardinal solutions. We introduced the definitions for separable objectives

and the neighbourhood between solutions. Also we stated the theoretical ideas of the algorithm and showed correctness. The algorithm is presented in pseudo-code but could be implemented in many solvers as a global constraint which we do not present here.

Already outlined is a generalisation of the algorithm such that the stored data can be updated from time to time if desired. Up to now there are no experimental results for the approach. So the next step is clearly to test it in a constraint solver environment and determine the outcome of applying this technique. Due to recent tests with weak symmetries we are very confident that this algorithm could considerably reduce the search.

Not investigated yet is the possibility to reduce the search effort for cardinal solutions by using a dominance criterion on the permutations to consider following the idea of Smith [9]. But using the idea we would have to change the way the permutations are investigated. This could mean that the approach may be faster in exhaustive search but might not be suitable for online optimisation as for example the application of the automated manufacturing. This has to be investigated in future.

References

[1] Roland Martin *The Challenge of Exploiting Weak Symmetries* In B. Hnich et al ed.: Lecture Notes in Computer Science, Volume 3978, 2006

[2] Peter Gregory *Almost–Symmetry in Planning* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005

[3] Alastair Donaldson *Partial Symmetry in Model Checking* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005

[4] Roland Martin *Approaches to Symmetry Breaking for Weak Symmetries* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005

[5] Warwick Harvey *Symmetric Relaxation Techniques for Constraint Programming* SymNet Workshop on Almost-Symmetry in Search, New Lanark, 2005

[6] Warwick Harvey *The Fully Social Golfer Problem* SymCon'03: Third International Workshop in Constraint Satisfaction Problems, Kinsale, Ireland, 2003

[7] Rico Gaudlitz *Optimization Algorithms for Complex Mounting Machines in PC Board Manufacturing* Diploma Thesis, Darmstadt University of Technology, 2004

[8] T. Fahle, S. Schamberger, and M. Sellmann *Symmetry Breaking* In CP 2001, pp. 225239, 2001

[9] Barbara M. Smith *Caching Search States in Permutation Problems* In CP 2005, pp.637-651, 2005

[10] Filippo Focacci, Paul Shaw *Pruning sub-optimal search branches using local search* In CPAIOR'02, pp. 181-189, 2002

A comparison of SBDS and Dynamic Lex Constraints

Jean-François Puget
ILOG
9 Avenue de Verdun
94253 Gentilly Cedex, France
puget@ilog.fr

Abstract

Many symmetry breaking methods have been proposed so far. Previous works have shown that these methods could be combined together under some conditions. We use a different angle : we compare the pruning power of two symmetry breaking methods. The first one is the rather classical SBDS method. The second one is the recently proposed dynamic lex constraints (DLC). We theoretically show that DLC prunes more nodes than SBDS if values are tried in increasing order. We also show experimentally that DLC can be more efficient than SBDS.

1 Introduction

Symmetries are mappings of a Constraint Satisfaction Problem (CSP) onto itself that preserve its structure as well as its solutions. If a CSP has some symmetry, then all symmetrical variants of every dead end encountered during the search may be explored before a solution can be found. Even if the problem is easy to solve, all symmetrical variants of a solution are also solutions, and listing all of them may just be impossible in practice. Breaking symmetry methods try to cure these issues.

SBDS is one of the most popular symmetry breaking methods. It has been proposed in [5], and further improved in [4]. It is a special case of the method proposed in [1]. This method adds conditional constraints during search in order to not revisit symmetrical variants of previously explored parts of the search tree.

Adding lexicographic constraints can break all symmetries[3]. However, it can be quite inefficient when the symmetry breaking constraints remove the solution that would have been found first by the search procedure. In order to overcome this issue, we have recently proposed in [12] to use a dynamic variable order (the one used during search) in the lexicographic constraints. This yields dynamic lexicographic constraints (DLC).

We performed an experimental comparison of these tow methods, and discovered that DLC resulted in a smaller search tree than SBDS. This paper presents an explanation of this fact. After some preliminaries in section 2, we recap the SBDS method in section 3. We then present DLC in section 4. A small example presented in section 5 shows that DLC can visit strictly less nodes than SBDS. Section 6 relates the two methods from a theoretical point of view. Section 7 contains some experimental results, and Section 8 contains our conclusions.

2 Preliminaries

We denote the set of integers ranging from 0 to $n-1$ by I^n.

2.1 CSP

We use standard notations for CSPs. A *constraint satisfaction problem* $\mathcal{P}$ (CSP) with n variables is a triple $\mathcal{P} = (\mathcal{V}, \mathcal{D}, \mathcal{C})$ where $\mathcal{V}$ is a finite set of variables $(v_i)_{i \in I^n}$, $\mathcal{D}$ a finite set of finite sets $dom(v_i)_{i \in I^n}$, and every constraint in $\mathcal{C}$ is a subset of the cross product $\bigotimes_{i \in I^n} dom(v_i)$. The set $dom(v_i)$ is called the *domain* of the variable v_i. Without loss of generality, we can assume that $dom(v_i) \subseteq I^k$ for some k.

The order in which variables appear in a (partial) assignment or in a solution is meaningful in the context of this paper. A *literal* is an equality $(x_j = a_j)$ where $a_j \in dom(x_j)$. An *assignment* is a sequence of literals such that the sequence of the variables in it is a permutation of the sequence of the variable v_i. A *partial assignment* is sub sequence of an assignment.

A *solution* to $(\mathcal{V}, \mathcal{D}, \mathcal{C})$ is an assignment that is consistent with every member of $\mathcal{C}$.

2.2 Symmetries

A *symmetry* is a bijection from literals to literals that map solutions to solutions. Our definition is similar to the semantic symmetries of [2]. We note l^g the application of the symmetry g to the literal l.

The symmetries of a CSP form a mathematical group. The inverse of a symmetry g is noted g^{-1}. The composition of a symmetry σ and a symmetry θ is $\sigma\theta$. The variable symmetries of a CSP form a sub group of its group of symmetries. The value symmetries of a CSP form a sub group of its group of symmetries.

We consider in this paper symmetries that are a combination of variable symmetries and value symmetries. Variable symmetries are defined by a permutation σ of I^n. Value symmetries are defined by a permutation θ of I^k. The effect of such symmetry on a literal is :

$$(v_i = a_i)^{\sigma\theta} = (v_{i^\sigma} = (a_i)^\theta) \quad (1)$$

If $S = (v_0 = a_0, v_1 = a_1, \ldots, v_{n-1} = a_{n-1})$ is a solution, then $(v_{0^\sigma} = (a_0)^\theta, (v_{1^\sigma} = (a_1)^\theta, \ldots, (v_{(n-1)^\sigma} = (a_(x-1))^\theta)$ is the solution $S^{\sigma\theta}$.

We have shown in [11] that the effect of value symmetries could be modeled by element constraints. An element constraint has the following form :

$$y = T[x]$$

where $T = [a_0, a_1, \ldots, a_{k-1}]$ is an array of integers, x and y are variables. The above element constraint is equivalent to :

$$y = a_x \wedge x \in I^k$$

i.e. it says that y is the x-th element of the array T. We will only consider injective element constraints, where the values appearing in the array T are pair wise distinct. In this case, the operational semantics of the element constraint is defined by the logical equivalence :

$$\forall i \in I^k,\ (x = i) \equiv (y = a_i)$$

We extend element constraints to sequences of variables. If $X = (v_i)_{i \in I^n}$ is a finite sequence of variables, then we define $T[X]$ as the application of an element constraint to each element of the sequence :

$$T[X] = (T[v_i])_{i \in I^n}$$

Element constraints can be used to describe applications of finite functions. For instance,

$$y = 3^x \wedge x \in I^4$$

is equivalently expressed through the following element constraint :

$$y = T[x] \wedge T = [1, 3, 9, 27]$$

Element constraint can also be used to represent the effect of value symmetries. Indeed, let θ be a value permutation corresponding to a value symmetry. By definition, any assignment of a value a to a variable x is transformed into the assignment of a^θ to x :

$$(x = a)^\theta = (x = a^\theta)$$

The permutation θ is represented by the array $T_\theta = [0^\theta, 1^\theta, \ldots, (k-1)^\theta]$. It defines a finite function that maps a to a^θ. The application of this function to x can be expressed by $T_\theta[x]$. We have represented the effect of the value symmetry by an element constraint.

More generally, if $(a_0, a_1, \ldots, a_{n-1})$ is the sequence of values taken by the variables $\mathcal{V} = (v_0, v_1, \ldots, v_{n-1})$, then $((a_0)^\theta, (a_1)^\theta, \ldots, (a_{n-1})^\theta)$ is the sequence of values taken by the variables $T_\theta[\mathcal{V}]$. Therefore, $S^\theta = T_\theta[S]$ for all solutions S.

Let us consider now the case where any symmetry is the composition $\sigma\theta$ of a variable permutation σ and a value permutation θ. The variable permutation σ is defined by a permutation of I^n. The value permutation is defined by a permutation of I^k.

If $(a_0, a_1, \ldots, a_{n-1})$ is the sequence of values taken by the variables $\mathcal{V} = (v_0, v_1, \ldots, v_{n-1})$, then $((a_{0^\sigma})^\theta, (a_{1^\sigma})^\theta, \ldots, (a_{(n-1)^\sigma})^\theta)$ is the sequence of values taken by the variables $T_\theta[\mathcal{V}^\sigma]$. Therefore, $S^{\sigma\theta} = T_\theta[S^\sigma]$ for all solutions S.

2.3 Search tree

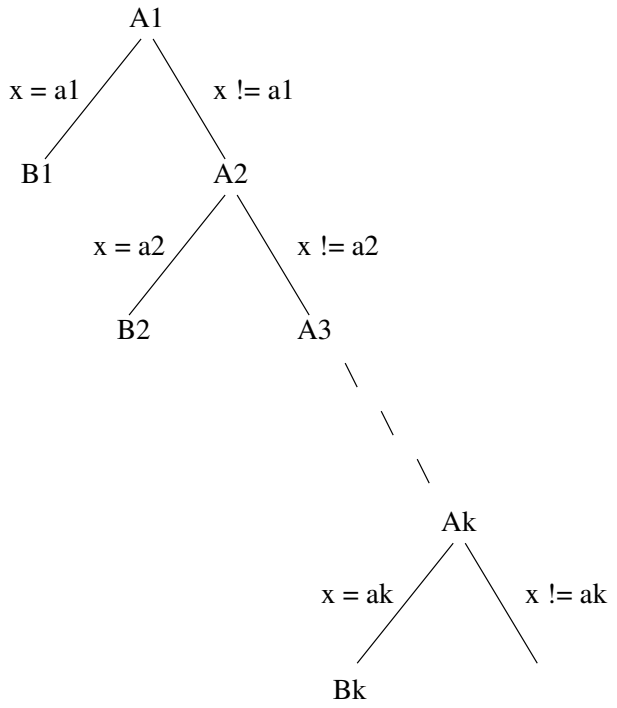


Figure 1: A tree search

We are interested in tree search methods (as opposed to local search methods for instance). Let us assume that the search tree is constructed as follows, see Fig. 1. In a given node A_1, then a variable x is selected. Then the all the values of the domain of x are tried before another variable can be selected. More precisely, if the domain of x contains the values $(a_1, \ldots, a_k)$ in increasing order, then A_1 has two children : one where $x = a_1$ is added, the other where $x \neq a_1$ is added. Let us call the former B_1 and the latter A_2. The node A_i (where $x \neq a_i$ is added) has two child nodes, B_i where $x = a_i$ is added, and A_{i+1} where $x \neq a_i$ is added. Note that in each node A_j, a_j is the minimal value in the domain of x :

$$x \geq a_j \quad (2)$$

This way of searching is quite common in CP systems and languages. It corresponds to what is called the "labeling" procedure in some CLP systems. It corresponds also to the "generate" procedure of ILOG Solver[6]. We restrict ourselves to this form of search because DLC apply to such search only.

Constraints can prune the tree : some nodes are inconsistent. These nodes have no children. Solutions are leaves of the search tree that are not inconsistent. Some constraint propagation algorithm may be applied at every node. It may result in some assignment of variables.

3 SBDS

SBDS posts conditional constraints during search. It assumes a depth first search of the tree. Let A_i be a non leaf node of the search tree (see Fig. 1), and let x be the variable selected for branching in this node. Let a_i be the value selected for x. Then the search proceeds with the sub tree rooted at B_i. Upon backtracking, the search proceeds with the sub tree rooted at A_{i+1}. For each symmetry $\sigma\theta$, SBDS adds the following constraint in A_{i+1}, where A stands for the assignment valid at the node A_{i+1} :

$$A \wedge A^{\sigma\theta} \wedge (x \neq a_i) \Rightarrow (x \neq a_i)^{\sigma\theta} \quad (3)$$

The constraint (3) is posted as a local constraint in A_{i+1}. It is valid only for the sub tree rooted at this node. In this node, both A and $(x \neq a_i)$ hold. Therefore, the constraint actually posted is :

$$A^{\sigma\theta} \Rightarrow (x \neq a_i)^{\sigma\theta} \quad (4)$$

The above is the original method described in [5] when the search tree is constructed as in section 2.3. Note that SBDS can be applied to more general trees. Various improvements to SBDS are presented in [4], but they will not be discussed in this paper.

Let us assume there are k variables fixed in the node A_i. Then, the partial assignment valid in this node is $A = (v_{i_0} = b_0, v_{i_1} = b_1, \ldots, v_{i_{k-1}} = b_{k-1})$. Let us rename v_{i_j} into x_j. Then, the partial assignment is $A = (x_0 = b_0, x_1 = b_1, \ldots, x_{k-1} = b_{k-1})$. Let us also rename x into x_k, and a_i into b_k.

Then, the constraint (3) is equivalent to :

$$(x_{0^\sigma} = (b_0)^\theta) \wedge \ldots \wedge (x_{(k-1)^\sigma} = (b_{k-1})^\theta) \Rightarrow (x_{k^\sigma} \neq (b_k)^\theta) \quad (5)$$

4 DLC

A very powerful symmetry breaking method has been proposed in [3]. The idea is to use a lexicographic order to compare solutions. Given two finite sequences $X = (x_0, x_1, \ldots, x_{n-1})$ and $Y = (y_0, y_1, \ldots, y_{n-1})$, we say that X is *lex smaller* than Y (denoted $X \preceq Y$) if, and only if :

$$\forall k \in I^n, \quad (x_0 = y_0 \wedge \ldots \wedge x_{k-1} = y_{k-1}) \Rightarrow x_k \leq y_k \quad (6)$$

We have introduced in [12] a new symmetry breaking method called dynamic lex constraints (DLC). These are lexicographic constraints such as in [3] with one major difference : the order in which variables appear in the lex constraints is not static. It is the order in which variables are selected during the search.

Then, DLC amounts to post the conditional constraint of (6) during the search. With the notations of Fig. 1., it posts at node A_1 the following constraints, where $A = (x_0 = b_0, \ldots x_{j-1} = b_{j-1})$ is the assignment valid in A_1 :

$$A \wedge (A)^{\sigma\theta} \Rightarrow (x_k \leq (x_{k^\sigma})^\theta) \quad (7)$$

This amounts to :

$$(x_{0^\sigma} = (b_0)^\theta) \wedge \ldots \wedge (x_{(j-1)^\sigma} = (b_{j-1})^\theta) \Rightarrow (x_k \leq T_\theta[x_{k^\sigma}]) \quad (8)$$

When $k^\sigma = k$, then the constraint $x_k \leq T_\theta[x_k$ can be propagated as follows. For any value a in the domain of x_k, if $T_\theta[a] < a$, then a can be removed from the domain of x_k. Indeed, $x_k = a$ would violate the constraint. This is somewhat reminiscent of the GE-tree method [13], as explained in [12].

5 An example

Both SBDS and DLC post conditional constraints. There are two differences. First, DLC posts constraints only in node A_1, whereas SBDS posts constraints in $A_2, \ldots, A_k$. Second, the right hand sides of the constraints posted by DLC are inequalities, whereas they are disequalities for SBDS.

Let us consider a simple example that clearly shows that DLC can prune more nodes. We consider a CSP with one variables $v \in \{0, 1\}$, with 2 values in its domain $(k =)$, and no constraint. We assume there is one value symmetry θ that swaps 0 and 1. It is defined by the array $T_\theta = [1, 0]$.

Let us look at the behavior of SBDS on this problem. At the root node, SBDS states no constraint. Then two children are constructed for this node, one where $v = 0$, the other where $v \neq 0$. Search proceeds with the first node. This node is a solution for the problem. Then search backtracks and visits the second node ($v \neq 0$). In this node, SBDS adds the following constraint :

$$v \neq T_\theta[0]$$

It is equivalent to :

$$v \neq 1$$

Then the node is inconsistent and the search stops here, after having visited 3 nodes.

Let us see what DLC would do on the same search. At the root node, it states :

$$v \leq T_\theta[v]$$

As explained in the previous section, the constraint is propagated as follows. Any value a such that $T_\theta[a] < a$ is

removed from the domain of v. Therefore, 1 is removed from the domain of v, since $T_\theta[1] = 0$.

Then at the root node v is set to 0 and the search stops with a solution.

On this tiny example, DLC has explored one node, whereas SBDS has explored 3 nodes.

We have seen that DLC can explore less nodes than SBDS. We will prove in the next section that the converse is not true. Indeed, any node pruned by SBDS would be pruned by DLC.

6 Comparing DLC and SBDS

We want to show that DLC will prune at least as many nodes as SBDS. First of all, DLC posts a constraint in A_1. This constraints can prune A_1 altogether, or it can prune some nodes in the sub tree rooted at B_1. Since SBDS posts no constraint on A_1, then no such pruning can occur. This was examplified in the previous section.

Let us look at the sub tree rooted at A_2. We want to show that the constraints posted by DLC are at least as strong as the ones posted by SBDS. Let us look at one of the node A_i, for $i > 1$. Let A be the assignment valid at this node. A is of the form $A = (x_0 = b_0, x_1 = b_1, \ldots, x_{k-1} = b_{k-1})$. The assignment valid at the node A_1 is a prefix of the assignment valid in A_i, i.e. it is of the form $A' = (x_0 = b_0, x_1 = b_1, \ldots, x_{j-1} = b_{j-1})$, with $j \leq k$.

SBDS posts (5) at node A_{i+1}.

DLC has posted (8) in A_1. Therefore, this constraint is valid is A_{i+1}.

In the node A_{i+1}, we have that $x_k > b_k$, because $b_k = a_i$ and because of (2). Therefore (8) implies the following in node A_{i+1} for all σ and for all θ :

$$(x_{0^\sigma} = (b_0)^\theta) \wedge \ldots \wedge (x_{(j-1)^\sigma} = (b_{j-1})^\theta) \Rightarrow (T_\theta[x_{k^\sigma}] > b_k) \quad (9)$$

Since it is posted for all θ, it is posted for all θ^{-1}:

$$(x_{0^\sigma} = (b_0)^\theta) \wedge \ldots \wedge (x_{(j-1)^\sigma} = (b_{j-1})^\theta) \Rightarrow (T_{\theta^{-1}}[x_{k^\sigma}] > b_k) \quad (10)$$

If, $x_{k^\sigma} = (b_k)^\theta$ then $T_{\theta^{-1}}[x_{k^\sigma}] = b_k$. Therefore, (10) implies :

$$(x_{0^\sigma} = (b_0)^\theta) \wedge \ldots \wedge (x_{(j-1)^\sigma} = (b_{j-1})^\theta) \Rightarrow (x_{k^\sigma} \neq (b_k)^\theta) \quad (11)$$

The left hand side of (11) is included in the left hand side of (5). Therefore, (11) implies (5). We have proved that the constraints posted by DLC imply the constraints posted by SBDS. This proves the following :

Theorem 1. *With the above notation and search procedure, any node pruned by SBDS would be pruned by DLC.*

The above proof may be easier to understand if we state constraints at a higher level. SBDS posts the following constraint at node A_i, for $i > 1$:

$$A \wedge A^{\sigma\theta} \wedge (x_k \neq a_i) \Rightarrow (x_{k^\sigma} \neq (a_i)^\theta) \quad (12)$$

DLC posts the following constraint at node A_1 :

$$A' \wedge (A')^{\sigma\theta} \Rightarrow (x_k \leq (x_{k^\sigma})^\theta) \quad (13)$$

(13) implies (12) because $A' \Rightarrow A$.

We will see in the next section that the reverse is not true : DLC may prune strictly more nodes than SBDS.

7 Experimental results

We reproduce here some of the experimental results reported in [12]. All experiments were run with ILOG Solver 6.3 [6] on a 1.4 GHz Dell D800 laptop running Windows XP.

A graph with m edges is *graceful* if there exists a labeling f of its vertices such that :

- $0 \leq f(i) \leq m$ for each vertex i,
- the set of values $f(i)$ are all different,
- the set of values $abs(f(i) - f(j))$ for every edge (i, j) are all different. They are a permutation of $(1, 2, \ldots, m)$.

A straightforward translation into a CSP exists where there is a variable x_i for each vertex i. These are hard CSPs introduced in [7]. They have been used as test bed for symmetry breaking methods, see for instance [8][11]. A more efficient CSP model for graceful graphs has recently been introduced in [15]. In this model, any symmetry of the graph induces a value symmetry. Together with this model, a clever search strategy is proposed. It is of the form described in section 2.3. It is shown in [15] that this search strategy clashes with lex constraints when they are used for breaking symmetries. For this reason, symmetries are broken using SBDS in [15].

We present in Table 1. and Table 2. results for various graceful graph problems. We use the implementation of [5] for SBDS, and our implementation for DLC [12]. Since all experiments use the same version of ILOG Solver and since they are run on the same computer we believe the comparison is fair. For each method we report the number of solutions found, the number of backtracks, and the time needed to solve the problem. Table 1. present results for finding all solutions (or prove there are none when the problem is unsatisfiable). Table 2. presents results for finding one solution when the problem is satisfiable.

These results show a significant increase of pruning from SBDS to dynamic lex constraints. They are consistent with Theorem 1. The improvement in running times aren't as large as the improvements in the number of backtracks.

8 Conclusion

We have compared two symmetry breaking methods, SBDS and DLC. Both methods post conditional constraints during search. Both methods do not interfere

Table 1: Results for computing all solutions for graceful graphs

Graph	SBDS			DLC		
	SOL	BT	sec.	SOL	BT	sec.
K3×P2	4	16	0.02	4	14	0.01
K4×P2	15	166	0.3	15	151	0.27
K5×P2	1	828	5.51	1	725	5.18
K6×P2	0	1839	43.2	0	1559	40.5
K7×P2	0	2437	149.4	0	1986	139.4

Table 2: Results for computing one solution for graceful graphs

Graph	No sym break		SBDS		dynamic lex	
	BT	sec.	BT	sec.	BT	sec.
K3×P2	0	0	5	0.01	5	0.01
K4×P2	16	0.08	12	0.05	12	0.05
K5×P2	2941	19.1	428	2.79	392	2.77

with the search ordering. We have shown that the constraints posted by DLC are stronger than the ones posted by SBDS when the search procedure is the standard labeling procedure. Experiments using graceful graphs are consistent with this theoretical analysis.

We intend to extend this comparison to other symmetry breaking methods such as SBDD and STAB. We think that such comparison might provide new insights into symmetry breaking, possibly yielding to the discovery of better symmetry breaking methods.

The comparison we did only considered the original SBDS method. It would be interesting to perform the same analysis with improved SBDS methods such as [4].

Last, DLC are currently limited to search where all values for a given variable are tried before another variable can be selected. SBDS is applicable to more general search procedures. It would be interesting to generalize DLC to general search procedures.

References

[1] Backofen, R., Will, S.: "Excluding Symmetries in Constraint Based Search" Proceedings of CP'99 (1999).

[2] Cohen, D., Jeavons, P., Jefferson, C., Petrie, K., Smith, B.: "Symmetry Definitions for Constraint Satisfaction Problems" In *proceedings of CP 2005*, ed. Peter van Beek, pp. 17-31, Springer, LNCS3709, 2005.

[3] Crawford, J., Ginsberg, M., Luks E.M., Roy, A.: "Symmetry Breaking Predicates for Search Problems". In *proceedings of KR'96*, pp. 148-159.

[4] Gent, I.P., and Harvey, W., and Kelsey, T.: "Groups and Constraints: Symmetry Breaking During Search" *In proceedings of CP 2002*, pp. 415-430.

[5] Gent, I.P., and Smith, B.M.: "Symmetry Breaking During Search in Constraint Programming" In *proceedings of ECAI'2000*, pp. 599-603.

[6] ILOG: *ILOG Solver 6.3. User Manual* ILOG, S.A., Gentilly, France, July 2006.

[7] I. J. Lustig and J.-F. Puget. "Program Does Not Equal Program: Constraint Programming and Its Relationship to Mathematical Programming". INTERFACES, 31(6):29–53, 2001.

[8] Petrie, K., Smith, B.M. 2003. "Symmetry breaking in graceful graphs." In *proceedings of CP'03*, LNCS 2833, pp. 930-934, Springer Verlag, 2003.

[9] Petrie, K.: "Comparison of Symmetry Breaking Methods in Constraint Programming" In *proceedings of SymCon05*, the 5th International Workshop on Symmetry in Constraints, 2005.

[10] Puget J.-F. 2005c. "Breaking All Value Symmetries in Surjection Problems" In *proceedings of CP 05*, pp. 490-504, 2005.

[11] Puget J.-F. "An Efficient Way of Breaking Value Symmetries" To appear in *proceedings of AAAI 06*.

[12] Puget J.-F. "Dynamic Lex Constraints" To appear in *proceedings of CP 06*.

[13] Roney-Dougal C.M., Gent, I.P., Kelsey T., Linton S.: "Tractable symmetry breaking using restricted search trees" In *proceedings of ECAI'04*.

[14] Seress, A. 2003. *Permutation Group Algorithms* Cambrige University Press, 2003.

[15] Smith, B.: "Constraint Programming Models for Graceful Graphs", To appear in *Proceedings of CP 06*