Using Symmetry of Global Constraints to Speed up the Resolution of Constraint Satisfaction Problems

Pierre Roy[1] and **François Pachet** [2]

Abstract. Symmetry in constraint satisfaction problems (CSP) can be used to either compute only a subset of the total solution set, or to prune branches of the search tree. However, detecting symmetry in general is a difficult task.

In this paper, we address the problem of detecting and exploiting a particular class of symmetry called *intensional permutability*, which is based on the notion of interchangeability between variables and can be detected with a very small overhead. This kind of symmetry is detected by collecting information on symmetrical properties of individual constraints. This method works particularly well on problems designed using global constraints.

We show how intensional permutability dramatically reduces the search tree for some problems. We propose a simple method to exploit it, which can be implemented as a lightweight extension to most resolution algorithms based on backtracking. We illustrate the method on several symmetrical problems, such as a classical layout problem and the pigeonhole problem, stated with a global constraint. Finally, we extend the method to symmetries involving groups of variables.

1 INTRODUCTION

Constraint satisfaction is a powerful paradigm for stating and solving complex combinatorial problems. The constraint satisfaction formalism was primarily designed as an algorithmic discipline [1] in which a problem is stated by characterizing *a priori* what is a solution. The resolution being undertaken by a general enumeration algorithm based on a tree search procedure [2] augmented with constraint propagation mechanisms [3, 4, 1]. This formalism was intended to be efficient, general-purpose and declarative as well.

However, the original formalism handles only binary constraints defined in extension, that is by the list of consistent instantiations of the variables it involves. These two limitations are not essential because 1) in the context of finite-domain problems, any constraint can be stated in extension and 2) any constraint can be fairly represented by a set of binary constraints.

Practically, handling constraints defined in extension leads to bloated problems and inefficient constraint propagation mechanisms. Moreover, most constraint problems are difficult to state with binary constraints because representing relations linking more than two variables requires to introduce several intermediate binary relations. Besides, lots of additional constraints and variables are needed, which increases the size of the problem, thus the complexity of the resolution.

Therefore, non-binary constraints, defined by formulas, are keys to reaching expected efficiency and declarativity for constraint satisfaction systems.

First, non-binary constraints can be used to speed up the resolution. Indeed, global filtering methods can be defined for non-binary constraints, which perform arc consistency, or any similar property, efficiently. This is e.g. the case complex constraints such as the global difference constraint [Régin 94] or global cardinality constraints [Régin 96], but also for standard linear constraints.

Second, using non-binary constraints allows to improve the declarative aspect of constraint satisfaction. Indeed, using non-binary constraints prevents the user from the task of decomposing the constraints of its problem in a set of binary constraints. For instance, consider n variables that are required to take different values. Using binary constraints, one would need to state $n.(n - 1) / 2$ binary difference constraints, while a single global difference constraint can do the job more efficiently [Régin 94].

There are a few more arguments in favor of non-binary constraints. One of them is that many complex well-known properties, coming from graph theory or operation research, can be expressed by non-binary constraints. Consider for instance the cycle constraint provided by CHIP [5], which can be used to state *complex* relations between variables straightforwardly. Moreover, this constraint is efficiently handled by specific filtering algorithms.

Another argument is that non-binary constraints are keys to implement sophisticated resolution mechanisms such as formal reasoning, as successfully experimented by J.-L. Laurière in the ALICE system [Laurière 78].

Despite the sophisticated techniques developed so far, many constraint satisfaction problems remain difficult to solve, and this is not surprising because the general constraint satisfaction problem is shown to be NP-complete.

However, even simple problems are out of reach for standard constraint-based resolution techniques. This is for instance the case of problems made up of several similar sub-problems. Many works were devoted to finding techniques adapted to such cases. For instance, research focused on finding better heuristics, decomposing problems [6, 7], considering the topology of the problem [8], or representing domain-specific knowledge (e.g. formal reasoning to solve numerical problems [9]).

There is another kind of problems that is surprisingly difficult to solve by standard constraint satisfaction techniques: *symmetrical* problems, whose paradigm is the famous pigeonhole problem [Puget 93].

It was already shown that symmetries could be used to speed up the search. For example, works of [10, 11] focus on symmetries inside the domains of variables while other works, e.g. [Crawford et al. 96] and [Puget 93], treats the case of symmetries between variables (see Section 2 below). They propose to introduce additional constraints to break the symmetry of the problem, thus reducing the complexity of the resolution.

In this paper, we focus on symmetry occurring between variables. First, we show that intrinsic properties of constraints can

[1] LIP6, Boîte 169, 4 Place Jussieu, 75 252 Paris Cedex 05, France. e-mail: roy@poleia.lip6.fr

[2] SONY CSL-Paris, 6 rue Amyot, 75 005 Paris, France. e-mail: pachet@csl.sony.fr

be used to detect symmetries. This detection is based on the exploitation of properties of non-binary constraints. Second, we show that, once detected, symmetries can be exploited straightforwardly by modifying the resolution algorithm. Moreover, this modification is shown compatible with every resolution algorithm based on the *backtracking + propagation* scheme. Finally, we extend our method to complex symmetries, supported by groups of constrained variables.

2 SYMMETRICAL CSPS

Many constraint satisfaction problems are *symmetrical*, in the sense that several solutions are equivalent with respect to permutations of some of their elements. Depending on what elements are permutable, different kinds of symmetries can be exhibited. We review below two of them: *symmetry on values* and on *variables*.

Symmetry on values occurs when several values in the domain of a given variable are interchangeable. For instance, consider the formulation of the queen problem where each queen is a variable whose domain contains all the squares of the chessboard. A 90-degree rotation of the chessboard leaves the problem unchanged. This rotation corresponds to the systematic permutation of square *(i,j)* with square *(j,i)*, which maps every solution onto another one. A survey of this kind of symmetry can be found in [11, 10].

Symmetry on variables corresponds to the situation where several variables play the same role. In this case, permuting similar variables maps every solution onto another one. A classical example is the pigeonhole problem, i.e. placing n pigeons in n-1 holes, with the constraint that one hole accommodates at most one pigeon. Consider the formulation of the problem with one variable for each pigeon, whose domain is the set of holes, and a single n-ary difference constraint. In the context of classical enumeration techniques, the complexity of this problem is in $O(n!)$, and even small problem instances require a huge computation time. Of course, a human would detect immediately that the problem has no solution, using some kind of commonsense knowledge: one cannot find an injection from a bigger set into a smaller set. This kind of inference is, in general, out of reach for classical constraint solvers. In this problem, there is a symmetry on variables, since pigeons are indistinguishable. This property can be used during the search to avoid repeatedly visiting “equivalent” parts of the search tree.

Another example is the n-queen problem, in which permuting any two variables leaves the problem unchanged. In this case, symmetry can be exploited to compute only a subset of the whole solution set. Table 1 below illustrates the influence of a symmetry detection on these two problems:

Table 1. Influence of the symmetry exploitation on the size of the search tree, on the number of solutions, and on the resolution time

Problem	No symmetry detection		Symmetry detection	
	Tree size (solutions)	**CPU (sec)**	**Tree size (solutions)**	**CPU (sec)**
6 pigeons	119 (0)	0.213	15 (0)	0.042
7 pigeons	719 (0)	1.031	31 (0)	0.064
8 pigeons	5,039 (0)	7.619	63 (0)	0.102
9 pigeons	40,319 (0)	61.902	127 (0)	0.228
5 queens	3,263 (1200)	13	300 (10)	0.9

Notice that the two types of symmetry are distinct. For instance, in the queen problem, symmetry on values represents a geometrical property of the chessboard while symmetry on variables represents the fact that queens are indistinguishable. The latter is independent of the topology of the chessboard.

This paper addresses the problem of detecting and exploiting automatically symmetry holding on variables. After a review of the main works in this area, we define a particular class of symmetry, based on a notion of permutability of variables, called *intensional permutability*.

Intensional permutability is a relation between variables that is computed from properties of the constraints. This property highly depends on the nature of the constraints of the problem, especially on their arity. More precisely, we will show that the use of global (non-binary) constraints is a key to detecting symmetry on variables.

We propose a simple scheme to exploit this symmetry, which can be implemented as a lightweight extension to most resolution algorithms. This method proves efficient on a whole class of problems, such as classical layout problem or the pigeonhole problem, and it generates a negligible overhead. Finally, we extend the method to symmetry involving composite structures.

3 STATE OF THE ART

The main result on symmetry on variables, presented by Puget [12], is outlined in this section.

Definition 1 CSP. *Let E be a finite set. A* CSP *P on E consists of the following elements:*

- A set $V=\{v_1, \ldots, v_n\}$; the v_i are the variables of P

- A mapping $D : V \rightarrow P(E)$; $D(v_i)$ is the domain of v_i. *We call D the Cartesian product of the domains: $D=D(v_1)\times\ldots\times D(v_n)$*

- A finite set C. Elements of C are the constraints

- A mapping $r: C \rightarrow P(D)$; $\forall c \in C$; r(c) is the set of tuples satisfying c, and is called the extension *of c*

In this definition, it is implicitly considered that every constraint involves all the variables of *P*. In the rest of this paper, the expression “the variables of constraint *c*” denotes the variables actually involved in the constraint.

Definitions 2 Instantiation and solution. *An* instantiation *of the variables is any $s \in D$. A* solution s *of P is an instantiation such that $s \in r(c)$ for every $c \in C$. (i.e. s satisfies all the constraints) The solution set of P will be denoted by S(P) or simply S.*

Let Π be the permutation group of the finite set $\{1,\ldots,n\}$. There is a canonical left-action of Π on the set D of the instantiations of the variables defined by:

$$\forall(\sigma,s) \in \Pi \times D\ ;\ \sigma.s = (s_{\sigma(1)}, s_{\sigma(2)}, \ldots, s_{\sigma(n)}).$$

Definition 3 Consistent permutation. *A* consistent permutation *is a permutation that maps any solution onto another one.*

The set of consistent permutations, equipped with the law coming from Π is a subgroup Γ of Π. By definition of Γ, S is stable for the action of Γ. Thus, Γ defines an equivalence relation[3] $\equiv$ on the solutions set S. When $\Sigma = \emptyset$, P is said to be *symmetrical.* The goal is therefore to compute $S/\equiv$ instead of S, in order to avoid computing equivalent solutions.

The central result of [12] is that, for any symmetrical CSP *P*, there exists a non-symmetrical CSP *P’*, deduced by adding new constraints to *P*, whose solution set is $S/\equiv$. However, deducing *P’* from *P* is expected to be as difficult as solving *P*. This remark suggests that the notion of consistent permutation is too general.

[3] When G acts on set E, the relation R defined by: $\forall(x,y) \in E \times E\ ;\ x\,R\,y$ *iff* $\exists g \in G$ *such that* $g.x = y$, is an equivalence relation

4 INTENSIONAL PERMUTABILITY

In this section, we define a stronger, thus less general, notion of symmetry on variables: symmetry that can be deduced from intrinsic properties of constraints, instead of being computed from their extensions. We do not regard the extension of a constraint as intrinsic since it depends on the domains of its variables.

Intuitively, two variables of a CSP are intensionally permutable if they play the same role for every constraint of the problem.

4.1 Preliminary Remarks

In the original formalism of CSP [1] (see definition above), a constraint is defined by the set of its consistent tuples. This is the *extensional* definition of the constraint.

Constraints can also be defined *intensionally*, that is to say by a formula of satisfaction. Remark that in the context of finite-domain constraint satisfaction, the two definitions are equivalent.

The method we propose to detect symmetry on variables relies on two ideas. The first idea is to exploit symmetrical properties of *individual constraints* to deduce symmetry of the whole CSP. The second idea is that symmetrical properties of a constraint can be deduced *directly* from its intensional definition, without considering its extension.

Let us illustrate these two ideas on the *n*-ary difference constraint. This constraint states that *n* variables $v_1, v_2, \ldots, v_n$ have different values. In this constraint, all the variables play exactly the same role. This is an intrinsic property of the constraint, which holds independently of any problem instance. This property is called *intensional permutability* and is defined in Section 4.2.

Remark that the general difference constraint was addressed by [13], who proposed a *filtering* method to enforce arc consistency in polynomial time. In the case of the pigeonhole problem, this allows to prove that there is no solution in polynomial time. It is important to note that our purpose and the one of Régin are complementary, and can be used simultaneously.

Technically, such symmetry corresponds to a partition of the variable set into subsets of intensionally permutable variables. These subsets are called *intensional permutability classes* (IP-classes for short). IP-classes represent to the idea that permutable variables will undergo the same events, i.e. the same domain reductions, during the resolution of the problem. This is the case when the problem is solved using any enumeration algorithm based on the property of arc consistency. Here are examples of IP-classes for standard constraints:

- The *n*-ary difference constraint is symmetrical. All the variables of the constraint are in a single IP-class.
- Linear constraints are not symmetrical in general. For instance, consider the following linear equality, involving coefficients:

$$\sum \alpha_i . v_i = 10$$

There is one IP-Class for each different value of the α_i.

- Comparison constraint, v > w, has two IP-classes: {v} and {*w*}.

In all cases, the constrained variables not involved in the constraint are in an additional IP-class.

The IP-classes of all the constraints will be combined together to yield global IP-classes gathering permutable variables of the problem. The combination consists in intersecting the IP-classes of all constraints. Once computed, the IP-classes of the whole problem can be exploited “on the fly” during the resolution to avoid useless exploration, as described in Section 5.3.

We now formalize the notion of intensional permutability, and compare it with the theoretical consistency defined in Section 3.

4.2 Definitions

Each constraint of the problem can be seen as a sub-problem likely to be symmetrical. The he following concept embodies this idea:

Definition 4 Sub-CSP generated by a constraint. *Let c be a constraint in a problem P. The* sub-CSP of *P* generated by *c, noted P(c), is the following CSP: P(c) = (V,* D*, {c},* $r_{|\{c\}}$*).*

We now define the notion of *strongly permutability*:

Definitions 5 Strong permutability. *Two constrained variables, u and v, are said to be* strongly permutable *if, and only if, for every constraint c of P, the transposition* $\tau_{u,v}$ *is consistent for P(c). If u and v are strongly permutable, the corresponding transposition, namely* $\tau_{u,v}$*, is said, by extension, to be also* strongly permutable.

The group Σ generated by strongly permutable transpositions is a subgroup of *Γ* (defined in Section 3). By definition, checking the strong permutability of two variables, say *u* and *v*, requires the study of the extensions of all the constraints involving *u* and *v*, which is an expensive process! To avoid this process, we introduce the definition of intensional permutability, which is based on IP-classes, and compare it to strong permutability.

Definition 6 Intensional permutability. *Two variables are said to be intensionally permutable if, for every constraint c, they are in the same IP-class for c. The corresponding transposition, namely* $\tau_{u,v}$*, is also said to be* intensionally permutable.

This relation can be computed *a priori* for each variable *u*, by simply intersecting the IP-classes containing *u*, for all the constraints of the problem. However, the group *Ψ* generated by intensionally permutable transpositions of the variables does not act on *S*, because the domains are not taken into account. As a counter-example, consider the CSP defined by $u \neq v$, where *D(u)*={1, 2} and *D(v)*={1, 2, 3}. Although this constraint is symmetrical, the transposition $\tau_{u,v}$ maps solution (1, 3) onto (3, 1), which is not a solution (because 3 is not in the domain of *u*). ($\tau_{u,v}$ is not consistent in the sense of Puget.) Hence, *Ψ* does not act on *S*.

Considering domains is therefore necessary to ensure the soundness of our method. To do so it is enough to ensure that variables having different domains at the statement of the problem are not considered strongly permutable. A simple solution consists in adding one global dummy constraint whose IP-classes are the sets of variables having the same domain.

For instance, consider a CSP with three variables *u*, *v* and *w* whose domains are respectively {0, 1}, {0, 1} and {1, 2, 3}. In this case, we add a constraint whose IP-classes are {*u*, *v*} and {*w*}. Since the permutable variables will eventually be computed by intersecting all the IP-classes, *u* and *w* will not be permutable.

Thanks to this additional constraint, *Ψ* now acts on *S*, thus inducing an equivalence relation ≈ on *S*. Since computing *S*/≡ instead of *S* is out of reach, we will show ho to compute only *S*/≈. Moreover, since *Ψ* contains consistent transpositions, it is a subgroup of *Σ*, and therefore: $\Psi \subset \Sigma \subset \Gamma \subset \Pi$. In the next two sections, we show that in general, $\Sigma \neq \Gamma$ and that $\Psi \neq \Sigma$.

4.3 Strong Consistency vs. Consistency

The following problem shows that consistency, in the sense of [12], does not always imply strong-consistency; i.e. there are cases where $\Sigma \neq \Gamma$.

Consider the Ramsey problem (see **Figure 1**), i.e. coloring the edges of K_4 using at most two colors, in such a way that there is no monochrome triangle. For that problem, $\Sigma = \emptyset$, because for any transposition p, one can find at least one solution s such that $p.s$ is not solution. However, Γ is not empty since it permutation: (X→Y→Z→X; U→V→T→U).

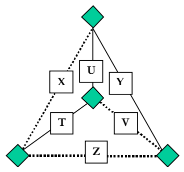

Figure 1. The Ramsey problem for K_4

4.4 Intensional vs. Strong Permutability

There are cases when two variables are strongly permutable although they are not intensionally permutable. Consider the cardinality constraint defined as follows: variable u represents the number of variables having value 1 in set $\{v\}$, with $D(u)=D(v)=\{0,1\}$. In this very particular case, u and v are strongly permutable for the constraint, since the following property holds:

$$(u=0 \Rightarrow v=0) \text{ and } (u=1 \Rightarrow v=1)$$

This constraint is equivalent to an equality constraint, which is symmetrical, while it has been stated as a cardinality constraint, which is not, thus leaving the permutability between u and v undetected. Therefore, $\Psi \neq \Sigma$.

Such a situation rarely occurs before the resolution. However, during the resolution, such permutability relations can appear, when several variables have had their domains reduced. Considering these situations would require the ability of dynamically reformulating constraints, which is beyond the scope of this paper.

5 EXPLOITING PERMUTABILITY

In this section, we present a method for exploiting permutability relations, both in order to compute fewer solutions (S/≈) and to speed up the search. Our method is designed to fit in classical constraint satisfaction techniques, i.e. techniques based on arc consistency and backtracking.

5.1 The Backtracking Procedure

The most widely used resolution algorithms (e.g. forward checking, full look-ahead) are based on a basic backtracking scheme that can be abstractly described using the two following procedures:

```
forward()
{
   if(crtVar == nil) crtVar = new_variable();
   if(crtVar == nil) solution_found();
   crtVal = new_value_for(crtVar);
   propagate_instantiation (crtVar ß crtVal);
   crtVar = nil;
   if(failure) backtrack()
   else forward()
}

backtrack()
{
   crtVar = old_variable();
   if(crtVar == nil) no_solution();
   remove_value_from_var(crtVal,crtVar);
   forward();
}
```

These procedures consist in instantiating progressively the variables. After each step, the current partial instantiation of the problem is checked against the constraints. If a constraint is violated, the procedure backtracks to the last instantiation.

5.2 Our Method

The method presented in [12] for exploiting consistency relations consists in adding constraints to the problem. This approach was also used for propositional satisfiability in [14]. As said in Section 3, automating such a transformation is out of reach.

Instead, we propose to modify directly the backtracking mechanism of the solver, by exploiting permutability relations after failures. Indeed, any choice of value, say x, for a variable u that led to a failure, can be safely removed from the domains of all variables permutable with u, as soon as the failure is detected (see Figure 2).

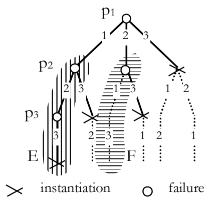

Figure 2. The search tree of the 4-pigeonhole problem. Dashed lines represent branches pruned by the method. Area F (horizontal hatches) is the image of E (vertical hatches) by transposition $\tau_{1,2}$. Therefore, once E has been explored, F can be safely pruned

This domain reduction is safe by virtue of the definition of permutability. This can be proven by refutation since permuting the values of u and v moves solutions from the left part of the search tree (area E) to the right part (area F), and vice versa. Let s be such a solution, in which $v = x$ (in the right part). Solution s', obtained from s by permuting the values of u and v would be located in area E. Since E contains no solution, this is a contradiction.

5.3 Implementation

The implementation of our method is straightforward, and consists in inserting a domain reduction in the backtracking procedure:

```
backtrack()
{
   crtVar = old_variable();
   if(crtVar == nil) no_solution();
   remove_value_from_var(crtVal,crtVar);
   for every V permutable with crtVar do
   {
      if(not(instantiated(V)))
         remove_value_from_var(crtVal,V);
   }
   forward();
}
```

The bold text is the sole modification of the algorithm to ensure that each permutability class will be visited only once during the search. This lightweight modification has no influence on the constraint propagation mechanism, or on the choice of the backtracking variable. It is therefore compatible with every algorithm based on the "consistency + backtracking" scheme (e.g. forward-checking, real full look-ahead and so forth, see [15]).

To be complete, our implementation requires two additional computations, which can be made before the resolution: the management of domains, by creating additional IP-classes (see Section 4.2), and the computation of the intersections of all IP-classes.

6 EVALUATION OF THE METHOD

In this section, we present our results on classical problems, and we discuss the current limitations of the method. The compromise obtained between the impact of the symmetry detection on the overall resolution and the overhead in efficiency and implementation is discussed in Section 8.

6.1.1 Two Little "Uniform" Problems

Using our method improves the resolution of the pigeonhole problem, since it detects right away that all the variables are permutable. The complexity of the resolution (i.e. proving unsatisfiability), is $O(2^n)$ instead of $O(n!)$. More precisely, using our method, the number of branches of the search tree developed is 2^{n-2}, instead of $(n-1)!$ For $n = 13$, $2^{n-2} = 2047$, while $(n-1)! = 40320$.

As for the 8-queen problem, using the formulation given in Section 1, the number of solution is 1,200. Using our method, only 10 solutions are computed. The size of the search tree passes from 3,263 to 300 branches (see Table 1).

6.1.2 A Mathematical Problem

Consider the following problem involving five variables *x, y, z, t, u* and *v* whose domain is {0, 1, …, 20}. The constraints are the following three equalities:

$$\begin{aligned} x + y + z + t &= 20 \\ x + y + z + u &= 20 \\ x + y + z + v &= 20 \end{aligned}$$

and a difference constraint between the five variables.

To prove the unsatisfiability of this problem, using a standard resolution scheme, 1,228 backtracks are required.

The constraints representing the three equalities are obviously symmetrical. If the difference constraint is represented by a set of ten binary difference constraints, after intersecting the IP-classes, no intensional permutability is detected.

On the contrary, if the difference constraint is represented by a single global constraint, the three variables *x*, *y* and *z* are intensionally permutable. In this case, the application of the method allows the unsatisfiability to be detected with only 199 backtracks. The resolution time decreases accordingly.

6.1.3 A Classical Layout Problem

A typical, and complex, symmetrical problem follows. Consider the decision problem of placing several rectangular plates of various dimensions into a rectangle area (see **Figure 3**). We assume that the plates and the rectangle area have integer height and width. Therefore, the problem can be stated as a finite-domain CSP. We consider a definition where the position of each plate in the rectangle is represented by a constrained variable. The domain of a variable is the set of possible positions for the corresponding plate. There is *a single constraint*, expressing that any two plates do not overlap. This constraint is symmetrical and may be stated as such. The system will therefore consider as permutable all plates having the same domain, i.e. having same dimensions.

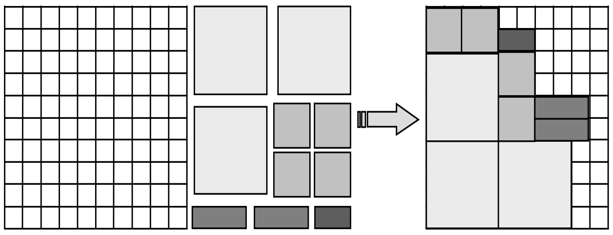

Figure 3. Putting plates in a rectangle. A solution is shown on the right. Permuting similar plates does not change the solution

For example, consider the instance Data4 described page 272 of [16], and illustrated by **Figure 3**. One has to place three 4×4 plates, four 2×2 plates, two 3×1 plates and one 2×1 plate on a 10×10 grid. Plates having same color are obviously permutable.

7 GENERALIZATION OF THE METHOD

In the past sections, we addressed symmetry occurring between single variables. We now generalize the method to symmetry holding on groups of variables.

When solving structured problems, classical enumeration algorithms are likely to apply repeatedly a same treatment to similar sub-problems. The essential idea is that a treatment that failed for a given sub-problem should not be applied to similar sub-problems.

Consider, for instance, the *optimization problem* corresponding to the preceding Layout Problem: placing the plates so that the height of the rectangle used is *minimized*. To state the problem, one has to define a special variable representing the highest position reached. The introduction of this variable changes the structure of the resulting CSP (see Figure 4), whose statement follows:

Variables $p_1,\ldots,p_n$ (the plates), whose domain is the set of possible position for each plate. Variables $h_1,\ldots,h_n$. h_i represents the highest position reached by plate p_i. Variable *H*, which represents the global highest position reached.

A non-overlapping constraint between the p_i's. A constraint between *H* and the h_i's. This constraint expresses that $H = max(h_1,\ldots,h_n)$. *n* constraints linking each p_i with each h_i, expressing that $h_i = p_i + y_i$, where y_i is the height of plate p_i. Figure 4 shows a graphical representation of the problem.

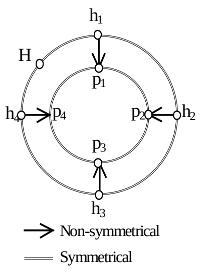

Figure 4. Graph of the optimization layout problem

The additional constraints break the relations of permutability since they are not symmetrical. However, similar plates remain *intuitively* indistinguishable. The inconsistency comes from the fact that now, the symmetry does not occur between the plate variables themselves, but between composite structures that are not explicitly represented as variables. These structures consist of one plate variable and its corresponding height variable together with the binary constraint linking them. In the next section, we extend our method for exploiting such higher-level symmetry.

7.1 Detecting Permutable Structures

Let us define the notion of *constrained structure* as a weak form of sub-CSP. A constrained structure *C* consists of a set of variables, denoted $\mathsf{V}(C)$, and a set of constraints (involving variables of $\mathsf{V}(C)$), denoted $\mathsf{C}(C)$. Given a constrained structure *C*, the constraints in $\mathsf{C}(C)$ are said to be *internal*; other constraints are *external*.

Given a CSP *P*, and a set of constrained structures of *P*, we build a *condensed* CSP *P'* such that the permutable classes of *P* correspond to the permutable constrained variables of *P'*. The condensed CSP *P'* is defined as follows:

- The variables of P' are the constrained structures of P and the remaining constrained variables of *P* (i.e. variables that do not belong to any constrained structure);
- The constraints of P' are constraints of P not internal to any constrained structure. For the sake of uniformity, they are reformulated to hold on the constrained structures instead of the original variables.

Two constrained structures C_1 and C_2 are *isomorphic* if the following one-to-one mappings do exist:

$$\Phi:\mathsf{V}(C_1) \longrightarrow \mathsf{V}(C_2) \text{ and } \Psi:\mathsf{C}(C_1) \longrightarrow \mathsf{C}(C_2)$$

such that for all $c \in \mathsf{C}(C_1)$ the following properties hold:

$$\text{(i) } \forall c \in \mathsf{C}(C_1);\ \Phi(\mathsf{V}(c)) = \Phi(\mathsf{V}(\Psi(c))),$$

$$\text{(ii) } \forall c \in \mathsf{C}(C_1);\ \Psi(c) \Leftrightarrow c\ ;\ (\text{i.e. } c \text{ and } \Psi(c) \text{ are similar})$$

$$\text{(iii) } \forall v \in \mathsf{V}(C_1);\ \mathsf{D}(v) = \mathsf{D}(\Phi(v)).$$

Two constrained structures of a CSP are *permutable* if 1) they are isomorphic and 2) the corresponding variables in the condensed CSP are permutable. Hence, permutable constrained structures are detected by applying the method to the condensed CSP.

For instance, in the plate problem, we can define constrained structures as the composition of one plate variable and its corresponding height variable together with the binary constraint linking them. These constrained structures are clearly pair-wise isomorphic. Moreover, the condensed CSP is symmetrical because the external constraints (non-overlapping and maximum height) are symmetrical for the constrained structures.

When considering large problems, detecting isomorphic constrained structures "from scratch" might be a difficult issue. However, we claim that in practice, isomorphic constrained structures are explicit in the design of the problem. This is the case, for instance, in the optimization layout problem above.

Once permutable constrained structures have been identified, we have to modify the resolution algorithm to exploit them. This is the purpose of the next section.

7.2 Modifying the Resolution Method

Permutable constrained structures are linked by an isomorphism defining a one-to-one mapping between their respective variables. The problem is that a variable and its image by the mapping are not actually permutable since constrained structures can be partially instantiated during the resolution.

Therefore, to prevent the method from omitting solutions, after a backtracking, the domain reduction will be propagated only to those variables that belong to constrained structures that have not yet been partially instantiated.

```
backtrack()
{
   crtVar = old_variable();
   if(crtVar == nil) no_solution();
   remove_value_from_var(crtVal,crtVar);
   for every V permutable with crtVar do
   {
      if(not(instantiated(V)))
      {
         if(V belong a non instantiated
            constrained structure)
         {
            remove_value_from_var(crtVal,V);
         }
      }
   }
   forward();
}
```

The correctness of this method is not reported here, for reasons of space limitations. We tested it on a set of instances of the layout problem [16], results are reported in Table 2. The solution reported here to exploit symmetry can be compared with the method proposed in [16], which requires a radically new design of the problem.

8 EFFICIENCY

Table 2 shows results on optimization layout problems solved with the BackTalk system [17, 18].

Table 2. CPU times for solving the optimization layout problem with and without the method. Solved instances are described in [16]

Instance	CPU not processed	CPU processed
Data #4	0.016 sec.	0.01 sec.
Data #5	67.846 sec.	3.039 sec.
Data #6	12.915 sec.	0.917 sec.
Data #9	0.807 sec.	0.44 sec.

The cost of the method has two components. First, the method requires the suppression of useless values after each backtracking. The cost for these suppressions is very low and is systematically compensated by the pruning effect gained. Second, the method requires the computation of the IP-classes. As written in Section 4.1, computing IP-classes consists in intersecting sets of variables, which can be done in polynomial time. Moreover, this process can be achieved *a priori*, i.e. at problem statement.

However, the method is limited to symmetry between pairs of permutable variables. As illustrated in Section 4.3, symmetries that are more complex are not addressed by the method. Detecting and exploiting efficiently such symmetry is a complex issue that would require a radically different approach.

9 CONCLUSION

We presented a method for detecting and exploiting a particular class of symmetry occurring in constraint satisfaction problems. Although it does not take into account all symmetry on variables, it drastically speeds up the resolution of non-trivial symmetrical problems.

The method exploits intrinsic symmetrical properties of individual constraints to decompose the set of variables into several subsets of permutable variables.

The method depends on the design of the problem, especially on the kind of constraints defined. More precisely, binary constraints break the symmetry of the problem because they decompose the problem into clusters containing at most two variables. Therefore, representing global constraints as such, instead of representing them by sets of binary constraints, is critical to ensure the success of the method.

The method is sound and complete, because all the permutability classes are visited at least once, and the entire solution set can be reconstituted from the permutability relations. Moreover, it allows a fast computation and a straightforward implementation, compatible with every enumeration algorithm based on arc consistency.

This method allows to get rid, in some sense, of highly symmetrical problems such as the pigeonholes, but is also proved useful on non-toy problems such as layout cutting. A finer evaluation of the frequency of real world problems yielding non-empty intensional permutability relations is under study.

We finally introduced an extension of the method to a broader spectrum of symmetry occurring between implicit structures of variables, which often occurs in structured problems.

REFERENCES

[1] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence,* 8(1), 99-118, 77

[2] R. Haralick and G. Elliot. Increasing Tree Search Efficiency for Constraint Satisfaction Problems. *Artificial Intelligence,* 14, 263-313, 80

[3] C. Bessière and J.-Ch. Régin. Arc Consistency for General Constraint Networks: Preliminary Results. *IJCAI-97,* Nagoya, Japan, 1, 398-404, Aug 97

[4] Y. Deville and P. Van Hentenryck. An Efficient Arc Consistency Algorithm for a Class of CSP Problems. *IJCAI-91,* 125-130, Chambéry, 91

[5] N. Beldiceanu and E. Contejean Introducing global constraints in CHIP *Journal of Mathematical and Computer Modelling,* 20 (12), 97-123, 94

[6] E. C. Freuder and P. D. Hubbe. Extracting Constraint Satisfaction Subproblems. *IJCAI-95,* 548-555, 95

[7] J. Larrosa. Merging Constraint Satisfaction Subproblems to Avoid Redundant Search. *IJCAI-97*, Nagoya, Japan, 1:424-429, Aug. 97

[8] R. J. Bayardo and D. P. Miranker. An Optimal Backtrack Algorithm for Tree-Structured Constraint Satisfaction Problems. *Artificial Intelligence,* 71, 159-181, 94

[9] J.-L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence,* 10(1), 29-127, 78

[10] E. C. Freuder and D. Sabin. Interchangeability Supports Abstraction and Reformulation for Constraint Satisfaction. *SARA-95,* 95

[11] B. Benhamou. Study of Symmetry in Constraint Satisfaction Problems. *PPCP-94*, Orcas Island, Washington, 246-254, May 94

[12] J.-F. Puget. On The Satisfiability of Symmetrical Constraint Satisfaction Problems. *ISMIS-93,* Norway, 93

[13] J.-Ch. Régin. A Filtering Algorithm for Constraints of Difference in CSPs. *Proceedings of AAAI-94*, 362-367, Seattle, Washington, 94

[14] J. D. Crawford, M. Ginsberg, E. M. Luks and A. Roy. Symmetry-Breaking Predicates for Search Problems. *Knowledge Representation,* Cambridge, MA, Nov. 96

[15] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence,* 9, 268-299, 93

[16] ILOGSOLVER User's Manual, 257-286, Ilog, France, 96

[17] P. Roy and F. Pachet. Reifying Constraint Satisfaction in Smalltalk. *Journal of Object-Oriented Programming,* 10(4), 51-63, June-July 97

[18] J.-Ch. Régin, Generalized Arc Consistency for Global Cardinality Constraint. *IJCAI-96*, Portland, Oregon, 1996.