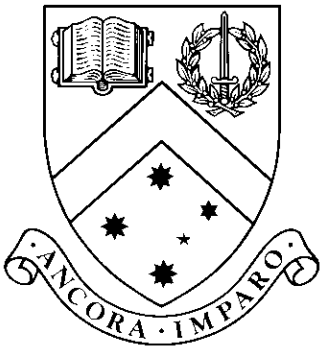
Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming

by

Christopher David Mears, BSoftEng (Hons)



Thesis

Submitted by Christopher David Mears

for fulfillment of the Requirements for the Degree of

Doctor of Philosophy (0190)

Supervisors:

A/Prof. Maria Garcia de la Banda

Prof. Mark Wallace

Clayton School of Information Technology
Monash University

October, 2009

Contents

List of Tables

List of Figures

Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming

Christopher David Mears, BSoftEng (Hons)
cmears@infotech.monash.edu.au
Monash University, 2009

Supervisor: A/Prof. Maria Garcia de la Banda
Prof. Mark Wallace

Abstract

Constraint satisfaction and optimisation problems occur frequently in industry and are usually computationally expensive to solve. Constraint programming is a technique for solving these difficult problems using specialised algorithms and search heuristics. The presence of symmetries in constraint problems provides an opportunity to reduce the computational effort required to solve these problems. To exploit symmetries, a problem must first be analysed to determine what symmetries are present and then the search algorithm must be modified to use the known symmetries.

In this thesis we provide contributions to both the research areas of automatically detecting symmetries and of exploiting them to improve search performance. The automatic detection of symmetries in constraint problems has been studied for some time, but existing methods can handle realistically-sized problems only by sacrificing the number and kinds of symmetry they can find. We contribute to this area in two parts. First, we present a method of automatically detecting the symmetries of a constraint problem that is capable of finding many small problems in a practical amount of time, and prove its correctness. Second, we use this method as the foundation of a framework for finding the symmetries in entire classes of problems. Symmetry detection on classes of problems has been little studied; our approach greatly improves the practicality of automatic symmetry detection as the symmetries found apply to many problem instances, small and large.

The exploitation of symmetries for improving search has been the subject of research for many years, but there is yet to be a method that is easy to use and that gives good performance under different search heuristics. We present a method of symmetry breaking, Lightweight Dynamic Symmetry Breaking (LDSB), that seeks to fill this void. We describe how LDSB focuses on common symmetries to maximise performance, and show experimentally that it performs consistently and is competitive with other dynamic symmetry breaking methods. Finally, we show how LDSB can be extended to much more general search techniques.

When considered together, these contributions form the components of an automatic system for detecting the symmetries of a problem class, and exploiting those symmetries when solving different instances of that class.

Automatic Symmetry Detection and Dynamic Symmetry Breaking for Constraint Programming

Declaration

I declare that this thesis is my own work and has not been submitted in any form for another degree or diploma at any university or other institute of tertiary education. Information derived from the published and unpublished work of others has been acknowledged in the text and a list of references is given.

Christopher David Mears

Acknowledgments

I owe thanks to the many people who have supported me during my PhD and before. I thank my supervisors, Maria and Mark, who provided advice, instruction, ideas, criticism, collaboration and friendship. Despite their busy schedules, their doors were always open. Maria also provided employment, and her dedication is inspirational and exemplary. I wish to thank also our collaborator, Bart Demoen, who not only did good work with us but also gave sage advice.

I thank my other colleagues at Monash University, with whom I shared many interesting discussions, particularly over lunch. They are too numerous to list here, but I wish to mention in no particular order Cagatay, Marc, Peter, Mauro, Peter, Kim, Bernd, Sarah, Michael, Cameron, Dhananjay, Chris, Tim, Daniel, Simone, Nathan, Reza, Kerri, Karen, Robyn, Joachim, Wenkai, Ranga and Pramudi.

I would like to thank those who gave assistance along the way, by pointing me in the right direction or answering my questions. This includes, but certainly is not limited to, Peter Stuckey, Toby Walsh, Joachim Schimpf, Mark Brown, Guido Tack, Mikael Lagerkvist, Claude-Guy Quimper, Justin Yip, Angelika Kimmig and Sebastian Brand. In particular I would like to thank Peter Baumgartner for his generous assistance in theorem proving, and Blair Bethwaite for helping to run experiments.

Finally, I thank those who have truly made my work possible: my family and friends. It is to you that I am most grateful.

Christopher David Mears

Monash University
October 2009

Chapter 1

Introduction

Combinatorial problems require the calculation of a particular combination of features that satisfy a set of constraints. Such problems occur frequently in logistical and industrial settings in areas as important and varied as environmental preservation, hospital management, drug design and resource allocation to optimise services and minimise costs. As an illustrative example, consider a simplified aircraft assignment problem whose aim is to allocate aircraft to the flights serviced by an airline company. In this case, the features are the flights to be covered and the fleet of aircraft available for use, while the constraints to be satisfied may relate to the capacity of the craft, regulations on aircraft operation times, the availability of crew at given times, and so on. Often the problems demand not merely an assignment that satisfies the constraints, but an assignment of high quality, where quality may be measured by its agreement with, for example, crew preferences, schedule flexibility, or maximised profit. The difference between a low and a high quality solution for a given problem might be measured in units as important as the number of millions of dollars lost, the amount of water waster or even the number of species made extinct.

Combinatorial problems are usually specified by a collection of variables, each of which may take a value from a certain domain, and a set of constraints. In our airline example above, each flight would be a variable and each aircraft a value the variables may take. The values taken by the variables should satisfy the constraints – that is, they should form a solution to the problem – and possibly also optimise some objective function that measures the quality of the solution. Unfortunately, combinatorial problems are difficult to solve and for many problems the best known algorithm to solve the problem includes some form of enumeration. That is, in order to find which combination of values form a solution, one must simply try a large number of combinations until a satisfactory one is found. At worst, all combinations must be tested. This strategy is inherently unsuitable for large problems, because as the number of variables and values increases, the number of possible combinations increases exponentially.

To illustrate the nature of the difficulty, consider a possible search for a solution to our airline example, shown in Figure 1.1. The steps followed by the search form a tree, usually called a search tree, where parts on the left side of the tree are explored before those on the right side. The first step of the search is to assign one of, say, five aircraft for the first flight, called flight A. The next step is to assign a craft for flight B, then for flight

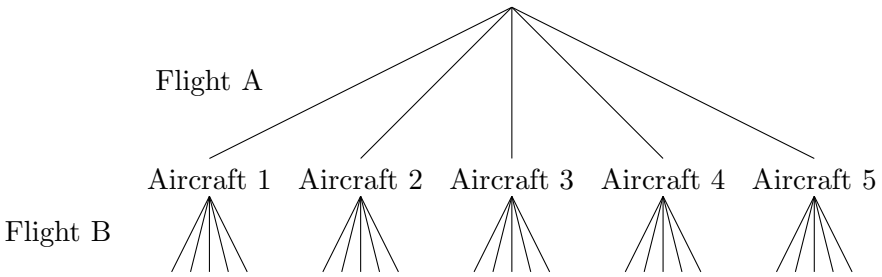


Figure 1.1: Part of a possible search tree for aircraft allocation.

C, and so on. At the bottom of the tree all flights have been allocated an aircraft, and each path from the top of the tree to the bottom represents a unique overall allocation. Whenever a subtree has been completely explored, the most recent decision is undone and the next untried case is examined. It can be seen that the number of overall allocations is exponential in size – adding just one more flight to the problem increases the number of allocations by a factor of five.

Constraint programming (Jaffar and Lassez, 1987; Marriott and Stuckey, 1998; Rossi et al., 2006) is a very effective technique for solving combinatorial problems. Arguably, two of constraint programming's strengths are its ability to significantly reduce (prune) the search tree, and the development of effective heuristics for guiding search choices. A region of the search tree can be pruned – that is, excluded from examination – if it can be determined that exploring the region would not lead to any solution. Pruning is achieved by examining the constraints at each node of the search tree and excluding the subtree under a given node if at least one constraint is known to be violated at that point. Search heuristics guide the decisions made by the search and can therefore have a significant impact on the time needed to solve the problem. The key decisions to specify a particular search heuristic are in which order to choose variables to assign values to, and in which order to try the values in each variable's domain. In the example shown in Figure 1.1, the variables (flights) are assigned in the order A, B, C, etc. and the values (aircraft) are tried in the order 1, 2, 3, etc. The variable and value ordering heuristics can be decided statically before the search, or chosen dynamically during the search.

Many combinatorial problems exhibit symmetry. A symmetry is a manipulation of an object that returns it to its original state. For example, as shown in Figure 1.2, rotating a rectangle 90 degrees around its centre changes its appearance while rotating it 180 degrees around its centre leaves the shape as it was originally. In the context of our airline example

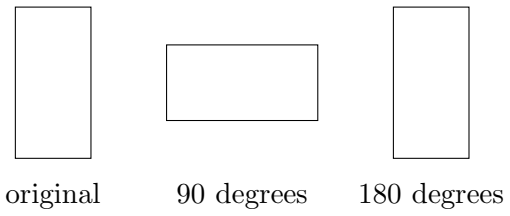


Figure 1.2: Rotational symmetry of a rectangle.

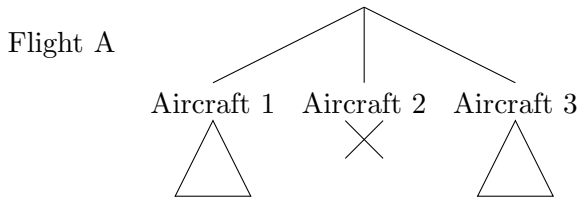


Figure 1.3: Partial search tree for aircraft allocation. If aircraft 1 and aircraft 2 are identical, the subtree under aircraft 2 can be ignored.

combinatorial problem, a symmetry might occur due to the airline's having two aircraft – say, aircraft 1 and 2 – of the same model that are in effect identical. Any allocation of aircraft to flights can be transformed into a different, but equivalent, allocation by exchanging the two identical aircraft. For example, the allocation A=1, B=2, C=5 is equivalent to the allocation A=2, B=1, C=5 if aircraft 1 and 2 are identical.

The presence of symmetry in a problem can provide opportunities for increased efficiency when the problem is solved using constraint programming. This is because the problem's symmetry causes some assignments of values to variables to be equivalent to other assignments. In our example, assume that aircraft 1 and aircraft 2 are identical. If the assignment of aircraft 1 to flight A is explored before considering aircraft 2, then there is no need to later consider assigning aircraft 2 to flight A (see Figure 1.3). To check every equivalent assignment would be redundant and possibly involve a great deal of pointless work.

There has been a significant amount of work done in defining the notion of symmetry in constraint programming (see e.g. (Cohen et al., 2005) for an overview), in devising algorithms for detecting symmetry (e.g. (Puget, 2005a)) and in exploiting symmetry (e.g. (Gent and Smith, 2000; Puget, 2002)). Our work in this thesis contributes to the latter two areas.

Research on detecting symmetries in constraint programs has covered many different approaches (e.g. (Backofen and Will, 1999; Freuder, 1991; Benhamou, 1994; Puget, 1993; Roy and Pachet, 1998; Meseguer and Torras, 2001; Ramani and Markov, 2004; Frisch et al., 2003b)), with the focus varying between speed and completeness. The task is to automatically find the symmetries, if any, of a given constraint program.

The most accurate and efficient method in this area (Puget, 2005a) is quite efficient and is able to detect all kinds of symmetry in a problem. However, its informal definition is incomplete and, as a result, it is not clear how it is applied to certain problems. In addition, much of its strength lies in its special treatment of certain constraints. As we will see later, while this can yield significant gains in efficiency, it can also result in a loss of accuracy.

Motivation 1. There is a need for a method that is accurate and able to work on any problem – regardless of the syntax used to express it – and that can run in a practical amount of time.

Even the best algorithms for detecting symmetries are limited either in the size of problems that they can handle, or in the kinds of symmetry they can find (such as only symmetries among the problem's variables, or only among its values). Although we present a new algorithm for symmetry detection that is highly accurate, it still does not scale well

to large problems. This is however not an issue for us, since our real aim is to use this method as the foundation of a new method that operates on constraint satisfaction models: i.e. on representations of the problem that contain abstractions of some of the data, rather than the actual values. In this way, the model represents many problem instances at once. For instance, our airline example can be generalised into a problem model by fixing the set of aircraft but leaving the exact number of variables – the flights – and their constraints unspecified. This reflects the situation of an airline that changes its flights daily but maintains a stable set of aircraft. In general, the data may describe the size of the problem or some other feature such as the capacity of a container, the cost of a resource, or the particular aircraft in a company's fleet. The model is instead defined in terms of the type of the data.

Just as an individual problem instance may have some symmetries, the model itself can also have symmetries. The symmetry among identical aircraft would in our example be a model symmetry, since no matter what set of flights is given as data, the symmetry among the aircraft is always present. Importantly, a symmetry that is present in the model must be present in every problem that is an instance of the model. Therefore, any symmetry present in the model is a symmetry of every possible instance of the model. As a result, a generally applicable symmetry detection method for constraint models would mitigate the efficiency problems of instance symmetry detection methods, as the symmetries could be found once for the model, and the results applied to many instances of the problem.

In contrast with symmetry detection for problem instances, there has been little work on automatically detecting symmetries in problem models. The primary reason for this is that the task is very difficult due to the complexity of the operations that a system needs to be able to abstract from a constraint program. This difficulty is clear when comparing the results of research in program analysis for logic programs with those for constraint logic programs. While many different frameworks have been proposed for logic programs and many analysers have proved successful for a range of useful optimisations, (e.g. (Muthukumar and Hermenegildo, 1991; Marriott and Søndergaard, 1993; Howe and King, 2003; Genaim and Codish, 2005; Puebla et al., 2005; Voets and De Schreye, 2009) and e.g. (Bossi and Deville, 1999)) very few frameworks and analysers have been proposed for constraint logic programs (e.g. (Marriott and Stuckey, 1994; Garcia de la Banda et al., 1996, 2000; Szilagyi et al., 2002)) and those that have been defined can only operate accurately when restricted to particular subsets of constraints.

We believe the inherent difficulty of analysing constraints is the reason why we know of only two methods for automatically detecting symmetry in constraint program models. Work on model symmetries by Van Hentenryck et al. (2005) focuses on finding the symmetries that are inherent in global constraints and composing them to find the overall symmetries of the model. This limits the applicability of the method to problems that are expressed using specific global constraints whose symmetries are well understood, and where the symmetries of the problem are preserved by the composition. Such an approach might be unable to detect symmetries that arise due to the conjunction of more than one constraint, as is common where global constraints are not used. Another approach to model symmetry by Roy and Pachet (1998) uses problem-specific methods of finding model symmetry and is therefore inherently not automatically applicable to arbitrary problems.

Motivation 2. There is a need for a symmetry detection method for models that operates on any kind of constraint problem without requiring particular constraints to be used, or imposing a large burden on the user. Finding the symmetries of a class of constraint problems at once greatly offsets the main drawback of accurate symmetry detection methods that operate on instances: the need for an impractical amount of computational effort for large problems. A method that operates on models needs to be run only once, and its results can be applied to all instances of the problem class.

Once the symmetries of a problem are known, they can be used to improve the efficiency of the search. The exploitation of symmetry is called symmetry breaking ((Puget, 1993; Crawford et al., 1996; Puget, 2005b,c; Flener et al., 2002a; Backofen and Will, 1999; Gent and Smith, 2000; Gent et al., 2005; Roney-Dougal et al., 2004; Gent et al., 2002, 2003; Focacci and Milano, 2001; Fahle et al., 2001; Sellmann and Van Hentenryck, 2005)) and can be broadly divided into two groups: static and dynamic. Static symmetry breaking methods add constraints to the problem before the search to exclude parts of the search space that are symmetrically redundant. Dynamic methods alter the search algorithm to make use of the symmetry as it progresses, deciding which redundant regions to eliminate during the search.

Each kind of symmetry breaking method has strengths and drawbacks. Static symmetry breaking methods have proved to be very efficient in many cases, but are sensitive to the variable and value ordering heuristics used by the search. In addition, the symmetry breaking constraints are not always easy to derive, and some powerful methods are available only when the symmetries have certain properties, such as when the symmetric variables are constrained to be distinct. Dynamic symmetry breaking methods are compatible with all search heuristics, but can impose a larger overhead on the search algorithm, especially when there are very many symmetries to be considered. For a new problem, it is not clear what kind of method – static or dynamic – will give the best performance, or even whether the best method might cause a decrease rather than an increase in speed.

Motivation 3. There is currently no method for symmetry breaking that can be used by default. Such a method must be easy to use, impose only a small and predictable overhead in most cases and often yield substantial improvements in search time. In particular, it should do so regardless of the particular search strategy used and the number of symmetries to be considered.

1.1 Thesis Outline and Contributions

In Chapter 2 we introduce the background and notation required for later chapters. The contributions presented in Chapter 3 arose from motivation 1: the need for an accurate symmetry detection method that works regardless of the syntax used to express the problem. To achieve this we first study the state of the art method by Puget (2005a), identify examples where this method is incorrect, and show how to modify the method to ensure correctness. We then define new forms of graph construction to represent a constraint satisfaction problem and its symmetries in a more accurate and less syntax-dependent way. We describe how to reduce the size of the resulting graph representations, by enforcing arc-consistency on the problem and by reducing the arity of the constraints. Finally, we evaluate an implementation of our method and compare it with Puget's method, showing

how our method finds all of the symmetries that Puget's can find, and does not require specially crafted representations of global constraints to do so.

The contributions presented in Chapter 4 arose from motivation 2: the need for a generally applicable symmetry detection method for constraint models. We present a radically novel approach that exploits the effectiveness of our graph-based detection method while avoiding its main drawback of inefficiency in large problems. The idea underpinning our approach is to find the symmetries of small instances of the problem and then to generalise them to the problem class. We also describe an implementation of the method that is capable of generalising the most commonly occurring symmetries by matching the instance symmetries against general symmetry patterns. We test our implementation on several varied benchmark problems and show that it finds almost all of the symmetries of the problems without having to examine large problem instances. In addition, we present two methods for automatically proving that a generalised instance symmetry is a symmetry of a model. Finally, we discuss how our approach, originally designed for symmetry, can be applied to the detection of other properties such as finding opportunities for caching. Like symmetry, caching has been shown to improve the efficiency of search in constraint programming (Smith, 2005; Garcia de la Banda and Stuckey, 2007).

The contributions presented in Chapter 5 arose from motivation 3: the need for a symmetry breaking method that performs well consistently and that can be used by default. We present a method for breaking symmetries during search called Lightweight Dynamic Symmetry Breaking, or LDSB, that is a formalisation of the shortcut SBDS method (Gent and Smith, 2000). The strengths of LDSB are that it adds little overhead to the search, it imposes a small burden on the user, and it often significantly improves search performance. LDSB focuses on symmetries that are common in practice and that can be represented simply and manipulated efficiently. In addition, we provide two publicly-available implementations of LDSB for the popular constraint programming platforms ECL^{i}PS^e and Gecode. Our experiments with LDSB show that despite its simplicity, it is competitive with other dynamic symmetry breaking methods and often surpasses them in performance.

The contributions presented in Chapter 6 arose from our experimentation with our symmetry breaking method, LDSB. We discuss how an extension of the popular SBDS method – and therefore LDSB – to more general branching constraints has pitfalls leading to incorrectness and describe how the extension can be done correctly for both SBDS and LDSB. We also examine the interaction between LDSB and constraint solvers and give minimum conditions on a solver to ensure that the symmetry breaking operates correctly. Finally, we show in a small experiment that our dynamic symmetry breaking method appears to be less sensitive to changes in search heuristics than a simple static symmetry breaking method and, thus, achieves more predictable behaviour.

When put together, the above contributions form an integrated, practical, implemented and widely available new system of handling symmetry in constraint programming, from automatic detection through to breaking. This and other conclusions are discussed in Chapter 7, where the contributions are summarised future work is discussed.

Chapter 2

Background

In this chapter we provide some of the background and notation required for the remainder of the thesis. While each chapter in this thesis contains a section explaining the particular items needed for that chapter, here we discuss terms and information common to all chapters. Citations given here are representative and not exhaustive.

2.1 Constraint Satisfaction and Optimisation Problems

A *constraint satisfaction problem* P is a triple (X, D, C) where X represents a set of variables, D a set of domains and C a set of constraints. Each variable $x_i \in X$ is associated with a domain $D_i \in D$ of potential values. In this thesis we will consider only finite domain variables, where every $D_i \in D$ is a finite set. In an abuse of notation, for convenience if $\forall D_i, D_j \in D : D_i = D_j$ – i.e. if all variables have the same domain – we will present the CSP in the form (X, D_i, C).

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

A constraint c is defined over a set of variables called its *scope* and denoted by $vars(c)$. A constraint c specifies a set of *allowed* assignments over $vars(c)$. An assignment over $vars(c)$ that is not allowed by c is *disallowed* by c. Let the *projection* of an assignment A over a set of variables V be defined as $\{l \in A : var(l) \in V\}$. An assignment A over $V \subseteq X$ *satisfies* constraint c if $vars(c) \subseteq V$ and the projection of A over $vars(c)$ is allowed by c. An assignment A over $V \subseteq X$ *violates* constraint c if $vars(c) \subseteq V$ and the projection of A over $vars(c)$ is disallowed by c. A *solution* of a CSP is a complete assignment which satisfies every constraint in C.

Example 1. The 8-queens problem requires the placement of 8 queens on an 8 by 8 chessboard such that no queen attacks another (see Section A.1 for a detailed discussion of the problem). This problem can be modelled as a CSP with one integer variable x_i for each queen i on the board so that each value, $d \in \{1, \ldots, 8\}$, represents the column

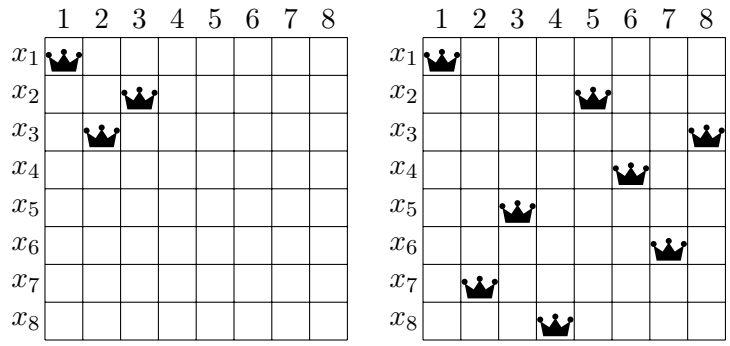


Figure 2.1: Assignments A_1 and A_2 for the 8-queens problem.

position of queen x_i in row i. The CSP is then
$(\{x_1, \ldots, x_8\}, \{1, \ldots, 8\}, \{\forall i, j \in \{1, \ldots, 8\}, i < j : x_i \neq x_j \wedge |x_i - x_j| \neq j - i\})$.

One literal of the problem is $x_2 = 3$, representing the queen in row 2 being placed in column 3. Consider an assignment A_1 (Figure 2.1, left) of the problem where $A_1 = \{x_1 = 1, x_2, = 3, x_3 = 2\}$. Assignment A_1 satisfies the constraint $x_1 \neq x_2$ because the projection of A_1 over $\{x_1, x_2\}$ is allowed by that constraint. Assignment A_1 violates the constraint $|x_3 - x_2| \neq 3 - 2$ because the projection of A_1 over $\{x_2, x_3\}$ is disallowed by that constraint. Assignment A_1 neither satisfies nor violates constraint $x_3 \neq x_4$ because the scope of the constraint is $V_1 = \{x_3, x_4\}$, assignment A_1 is defined over the set $V_2 = \{x_1, x_2, x_3\}$ and $V_1 \not\subseteq V_2$. Consider another assignment $A_2 = \{x_1 = 1, x_2 = 5, x_3 = 8, x_4 = 6, x_5 = 3, x_6 = 7, x_7 = 2, x_8 = 4\}$ (Figure 2.1, left). This assignment is defined over every variable of the problem, so it is a complete assignment. It also satisfies every constraint of the problem, so it is a solution of the problem. □

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

Example 2. The constraint $x_i \neq x_j$ used in the above 8-queens problem can be represented extensionally as the set $\{(1, 2), (1, 3), \ldots, (1, 8), (2, 1), (2, 3), \ldots\}$ of allowed assignments, or intensionally by the function $f(a, b) = (a \neq b)$. □

An extension of a constraint satisfaction problem is a constraint *optimisation* problem. In an optimisation problem, the task is to find an assignment that satisfies all constraints and that optimises (usually, minimises or maximises) some objective function. We focus only on constraint satisfaction problems here, although many of the topics covered apply to optimisation problems as well.

2.2 Constraint Programming

Constraint programming (Rossi et al., 2006; Marriott and Stuckey, 1998; Jaffar and Lassez, 1987) is a paradigm for solving constraint satisfaction and optimisation problems. The use of constraint programming to solve a problem requires three parts: a specification of the problem, a constraint solver, and a search algorithm. The recent trend in constraint programming is to use a CP modelling language to specify the problem (e.g. (Smolka, 1995; Van Hentenryck, 1999; Garcia de la Banda et al., 2006; Frisch et al., 2007; Flener et al., 2004; Van Hentenryck and Michel, 2005) Modelling languages for constraint programs vary and we do not discuss them in detail here. Instead, we focus on constraint solvers and search algorithms. For simplicity, in this thesis we will use simple mathematical notation rather than any particular modelling language.

2.2.1 Finite Domain Constraint Solvers

A constraint solver's task is to determine whether a CSP has a solution. Given a CSP P, a solver outputs either "yes", meaning there is at least one solution, "no", meaning there is no solution, or "maybe", meaning the solver has not yet determined with certainty whether there is a solution. The ways by which constraint solvers produce their answers vary, but in the case of CSPs where the variables have finite domains, most solvers operate by propagating constraints.

Constraint propagation is the process of "reducing" a problem by eliminating from the domains of the variables those values that can be determined never to participate in a solution. A CSP $P' = (X', D', C)$ is a *reduced* form of a CSP $P = (X, D, C)$ if $X = X' \wedge \forall D'_i \in D'.D'_i \subseteq D_i$ and the set of solutions of P is the set of solutions to P'. The values removed from any D_i ($D'_i \setminus D_i$) are said to have been *pruned*.

There are many ways in which a constraint solver can propagate constraints. One method of propagating a constraint is to achieve some form of consistency. Below we discuss node consistency, (generalised) arc consistency and bounds consistency.

A constraint c is *unary* if $vars(c)$ is a singleton set. Likewise, a constraint c is *binary* if $vars(c)$ has cardinality two. A problem is *node consistent* with respect to a unary constraint c where $vars(c) = \{x_i\}$ if d_i is a subset of the allowed assignments of c. In other words, every value in d_i satisfies constraint c. A problem is *arc consistent* (Waltz, 1975; Mackworth, 1977a) with respect to a binary constraint c where $vars(c) = \{x_i, x_j\}$ if for every value $a \in d_i$ there exists a value $b \in d_j$ such that the assignment $\{x_i = a, x_j = b\}$ is allowed by c. For a given value $a \in d_i$, the allowed assignment $\{x_i = a, x_j = b\}$ is called a *support* for $x_i = a$. The relationship among the variables induced by the notion of support is directional, in the sense that for a given constraint c where $vars(c) = \{x_i, x_j\}$, there may be a value for x_i that has no support while every value for x_j has a support.

A problem is called node (arc) consistent if it is node (arc) consistent with respect to every unary (binary) constraint in the problem.

Example 3. Consider the CSP $(\{x_1, x_2\}, \{1, 2, 3\}, \{x_1 < x_2\})$. It can be seen that x_1 can never take the value 3, because there is no allowed assignment for the constraint that contains $x_1 = 3$. Similarly, x_2 can never take the value 1 because there is no allowed

assignment for the constraint that contains $x_2 = 1$. The reduced, arc consistent version of this problem is the CSP $(\{x_1, x_2\}, \{\{1, 2\}, \{2, 3\}\}, \{x_1 < x_2\})$. □

Arc consistency can be extended to constraints whose scope contains more than two variables (Mackworth, 1977b; Freuder, 1978). A problem is *generalised arc consistent* with respect to a constraint c where $vars(c) = S$ if, for every variable $x_i \in S$ and every value $a \in d_i$, there exists an assignment A over S such that $(x_i = a) \in A$ and for every literal $(x_j = b) \in A$ it is true that $b \in d_j$. As for arc consistency, such an assignment A is called a support for $x_i = a$.

A naive way of propagating a constraint is to achieve generalised arc consistency by checking for a *support* for every value of every variable in the scope of the constraint, and removing from its domain any value that does not have a support. Given a constraint c, if every value of every variable in $vars(c)$ has a support, then the domains of the variables cannot be reduced any further by propagation of c. If the problem is generalised arc consistent with every constraint of the problem, then the problem is said to be generalised arc consistent.

When the values in the domains of the variables are numbers – or more generally, totally ordered – another form of consistency is possible. Instead of trying to find a support for every value in the domain of each variables, only the bounds – the minimum and maximum values – of each domain are tested. This form of consistency is often less expensive to achieve and, in many cases, leaves the domains in the same state as generalised arc consistency. In general, however, bounds consistency is weaker than generalised arc consistency; there are cases where generalised arc consistency can reduce the domains further than bounds consistency.

Example 4. Consider the CSP:

$$(\{x_1, \ldots, x_5\}, \{\{1, 2, 3, 4, 5\}, \{2, 3\}, \{2, 3\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\}\}, \{\mathit{all\_different}(\{x_1, \ldots, x_5\})\})$$

(where the *all_different* constraint ensures that the five variables take distinct values). To achieve bounds consistency the minimum and maximum values of each domain are checked to see if they have a support. As for generalised arc consistency, we consider $x_i = a$ to have a support for constraint c if there exists an assignment A over $vars(c)$ such that $(x_i = a) \in A$ and for every literal $(x_j = b) \in A$ we have that $b \in d_j$. $x_1 = 1$ has no support, so 1 is removed from d_1. The new minimum of d_1 is 2, which has no support and is removed. Likewise, the new minimum 3 is removed, and so is 4. $x_1 = 5$ has a support and remains the only value in d_1. After that, the minimum and maximum values of every variable have a support, and the reduced CSP is $(\{x_1, \ldots, x_5\}, \{\{5\}, \{2, 3\}, \{2, 3\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\}\}, \{\mathit{all\_different}(\{x_1, \ldots, x_5\})\})$. Generalised arc consistency can also remove the values 2 and 3 from the domains of x_4 and x_5. □

In general, propagation is incomplete. That is, it may leave a value v in the domain of a variable x even though the literal $x = v$ does not participate in a solution. This is because the constraints are considered separately and domain reductions that rely on their conjunction are not inferred.

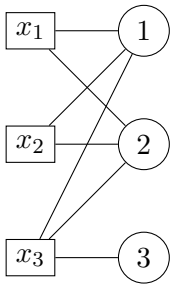


Figure 2.2: Bipartite matching problem constructed for the all-different constraint in the CSP $(\{x_1, x_2, x_3\}, \{\{1,2\}, \{1,2\}, \{1,2,3\}\}, \{all\_different(\{x_1, x_2, x_3\})\})$.

Example 5. Consider the CSP $(\{x_1, x_2, x_3\}, \{\{1,2\}, \{1,2\}, \{1,2,3\}\}, \{x_1 \neq x_2,\ x_1 \neq x_3,\ x_2 \neq x_3\})$. Enforcing generalised arc consistency does not alter the domains of the variables. However, the literal $x_3 = 1$ cannot participate in a solution and could be pruned. □

To this point we have considered mainly *primitive constraints*. While the precise definition of a primitive constraint is not clear, a constraint is usually considered primitive if it is supported directly by a constraint solver and cannot be decomposed into a combination of simpler constraints.

More recently, a different kind of constraint has arisen from the observation that there are certain patterns of constraints that occur frequently in constraint programs. For example, a set of variables may be constrained to take distinct values. This constraint can be represented by a conjunction of $x_i \neq x_j$, with one such constraint for each pair of variables in the set. However, as shown in Example 5, this may give inadequate propagation. A *global constraint* is a specialised constraint designed to capture a common pattern (see (Beldiceanu et al., 2007) for a library of global constraints). One global constraint that is very commonly used in constraint programming is the *all-different* constraint. This constraint takes as a parameter a set of variables ensures that every variable in that set takes a distinct value.

Example 6. Consider the CSP:
$(\{x_1, x_2, x_3\}, \{\{1,2\}, \{1,2\}, \{1,2,3\}\}, \{all\_different(\{x_1, x_2, x_3\})\})$
This problem has the same variables and domains as the CSP in Example 5, but uses a single global constraint instead of three $\neq$ constraints. □

It is possible to replace a global constraint with a set of non-global constraints that has the same meaning. However, global constraints are useful because they can often be propagated more effectively and/or more efficiently than the equivalent set of simpler constraints. For example, a propagation algorithm for the all-different constraint exists that prunes all possible domain values and runs in polynomial time (Régin, 1994). This algorithm is based on the construction of a bipartite graph representing the variables and values involved in the constraint and finding a maximum matching on the graph. For example, Figure 2.2 shows the matching constructed for the all-different constraint in Example 6.

In practice the algorithms used for constraint propagation and the level of consistency they achieve depend on the constraints involved. Usually node consistency is achieved for

unary constraints and at least bounds consistency is achieved for binary constraints. In the specific case of linear arithmetic constraints, such as $x_1 + x_2 + x_3 = x_4$, bounds consistency is often as strong as arc consistency, and the cost of achieving bounds consistency is much less than for generalised arc consistency. At a minimum, the propagation algorithm must be able to detect whether an assignment violates a constraint.

For a problem with more than one constraint, one possible method of propagation is to examine each constraint in turn, possibly more than once. In practice, this is achieved by representing each constraint by a propagator, i.e. a daemon that watches for changes to the domains of the variables in the constraint's scope and, upon such a change, is added to a queue to execute the consistency algorithm. What constitutes a change to a domain depends on the propagator itself; it may be a reduction in the size of the domain, a change in the minimum or maximum values of the domain, etc. The propagators in the queue are executed in turn until the queue becomes empty.

Example 7. Consider the CSP $(\{x_1, x_2, x_3\}, \{1, 2, 3\}, \{x_1 < x_2,\ x_2 < x_3\}$. Propagation proceeds as shown in the table below. At each line, the propagator for the constraint at the front of the queue is executed. Any change in the domain of a variable in the scope of a constraint may cause that constraint's propagator to be added to the queue.

d_1	d_2	d_3	queue
{1,2,3}	{1,2,3}	{1,2,3}	$x_1 < x_2,\ x_2 < x_3$
{1,2}	{2,3}	{1,2,3}	$x_2 < x_3$
{1,2}	{2}	{3}	$x_1 < x_2$
{1}	{2}	{3}	-

□

There are many solvers commonly used for finite domains, including Gecode (Gecode Team, 2006), the IC and FD solvers for ECL^{i}PS^e (Wallace et al., 1997), MINION (Gent et al., 2006), ILOG Solver (ILOG) and Comet (Van Hentenryck and Michel, 2005).

2.2.2 Search

Constraint propagation rarely reduces the domains of the variables of a problem to single values. In most cases, it is necessary to reduce the problem further by way of search, i.e. by exploring the different values in the domain of the variables. Here we will discuss one of the most common search mechanisms in constraint programming: a complete, backtracking tree search that branches by assigning values to variables. The search is said to be complete because every solution (if any) of the problem will be found, and if no solution is found then the problem is known not to have a solution.

Consider the CSP $P = (X, D, C)$; the search proceeds as follows (see Figure 2.3). First, the constraints in C are propagated, leading to a reduced problem $P' = (X, D', C)$. If the domain of any variable has become empty – i.e. if there exists any $d'_i \in D'$ such that $d'_i = \emptyset$ – then there must be no solution to P' and the search fails. If all variables have domains of size one – i.e. for all $d'_i \in D'$ the cardinality of d'_i is one – then the current domains D' represent a solution to the problem and the search returns that solution. If the search has failed, or if a solution is found but more solutions are sought, then the search backtracks to the nearest ancestor in the tree where there is a child yet unexplored, and select a different child to explore.

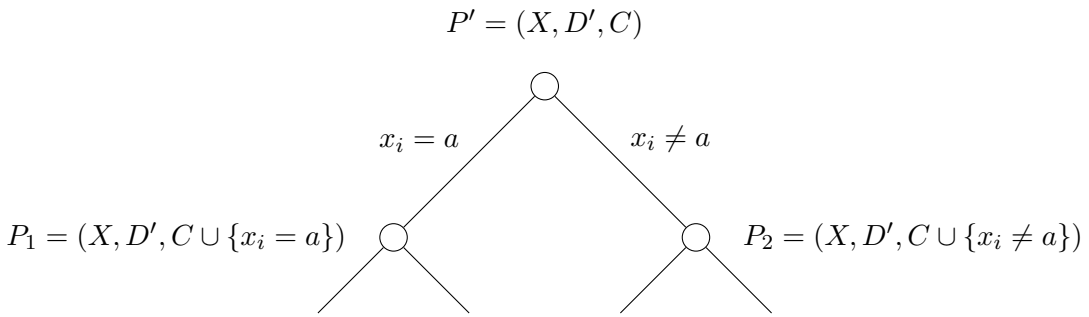


Figure 2.3: Depth-first search tree.

If the search has not failed and there remains any variable whose value is not yet fixed – whose domain has size at least two – then the search chooses such an unfixed variable x_i and a value a from the domain of x_i. The search then splits the problem into two subproblems $P_1 = (X, D', C \cup \{x_i = a\})$ and $P_2 = (X, D', C \cup \{x_i \neq a\})$ and recurses on subproblem P_1 until a solution is found. If no solution to P_1 is found then the search backtracks to P_2. If neither P_1 nor P_2 has a solution, then P has no solution.

Note that at each point of the search tree where the search branches, we must select a variable and a value on which to branch. The order in which variables and values are tried can have a large impact on the search performance. Variable and value orders can be fixed before the search begins, or chosen dynamically as the search progresses.

Most constraint programming platforms (e.g. Gecode (Gecode Team, 2006), ECL^{i}PS^e (Wallace et al., 1997), Minion (Gent et al., 2006), etc.) offer a few common, generic variable ordering heuristics such as input order and first fail, and a few value ordering heuristics such as ascending order, descending order or middle-out order. The variable ordering heuristics take as a parameter a sequence of variables to be assigned values by the search. The *input order* variable ordering assigns values to variables in the order that they are specified; e.g. if the variables are $[x_1, x_2, \ldots, x_n]$, then x_1 is tried first, then x_2, and so on. The *first fail* variable ordering selects the variable from the given sequence with the smallest domain size. This is calculated at the time of branching, according to the domains as they have been reduced by search and propagation. Alternatively, the user can specify a customised ordering of variables and values that is tailored the the specific problem to be solved.

The variable and value orderings can have a dramatic effect on how much of the search space needs to be explored in order to find a solution. A good heuristic is one that is more likely to guide the search towards a solution earlier in the search. For example, Figure 2.4 shows a search tree for a problem where no solution contains the literal $x_1 = 1$, but there is a solution where $x_1 = 2$. If the search was to try $x_1 = 2$ before $x_1 = 1$, then the solution would be found more quickly.

A feature of constraint programming is the ability to easily specify problem-specific search algorithms. In some cases a generic search is inadequate and the use of a customised strategy for dividing the problem into subproblems is vital for efficient resolution. For example, for the N-queens problem (see A.1) it is known that using the values in the middle of the domain first is an effective strategy. As another example, Puget and Smith

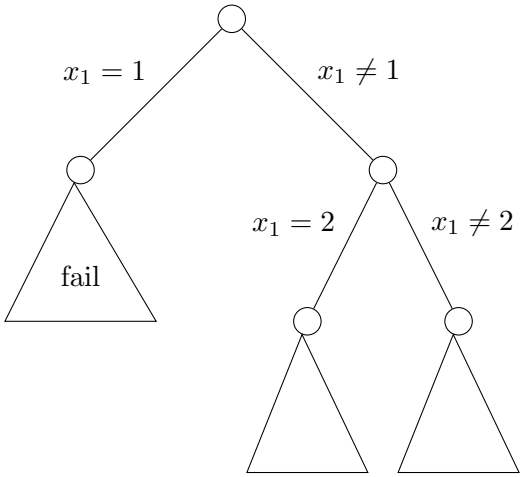


Figure 2.4: Search tree where $x_1 = 1$ is tried before $x_1 = 2$. There is no solution containing $x_1 = 1$.

(2006) show that a problem-specific variable ordering for the graceful graph labelling problem (see A.9 gives good performance.

2.3 CSP symmetry

A *solution symmetry* σ of a CSP is a permutation of its literals that preserves the set of solutions (Cohen et al., 2005). A symmetry can be extended to act on assignments by applying the symmetry to each literal in the assignment: $\sigma(A) = \{\sigma(l)|s \in A\}$. In other words, a symmetry is a bijection from literals to literals that maps solutions to solutions and consequently non-solutions to non-solutions. As a consequence, for any solution symmetry σ, assignment A is a solution if and only if $\sigma(A)$ is a solution. A *constraint symmetry* is a solution symmetry that preserves the constraints of the CSP. That is, the symmetry maps each constraint in the set of constraints C onto a constraint that is in C (possibly the same constraint).

Example 8. Consider the CSP $P = (X, D, C)$ where $X = \{x_1, x_2, x_3\}$, $D = \{1, 2, 3\}$ and $C = (x_1 + x_2 = 5) \wedge (x_3 < 3)$. One solution of this problem is $A = \{x_1 = 2, x_2 = 3, x_3 = 1\}$. Let us define the permutation of literals f such that $f(x_1 = i) = (x_2 = i)$, $f(x_2 = i) = (x_1 = i)$ and $f(x_3 = i) = (x_3 = i)$; i.e. f interchanges the literals involving x_1 and x_2. Permutation f is a solution symmetry of P because it maps solutions to solutions; for example, $f(A) = \{x_1 = 3, x_2 = 2, x_3 = 1\}$, which is also a solution of P. Now let us define the permutation g such that $g(x_1 = i) = (x_1 = i)$, $f(x_2 = i) = (x_3 = i)$ and $f(x_3 = i) = (x_2 = i)$; i.e. g interchanges the literals involving x_2 and x_3. Permutation g is not a solution symmetry of P because it maps solutions to non-solutions and vice versa; for example, $g(A) = \{x_1 = 2, x_2 = 1, x_3 = 3\}$, which is not a solution of P. □

The presence of symmetry in a constraint satisfaction problem can be used to reduce the search for the problem's solutions. This is because a symmetry causes some subtrees of the search space to be equivalent, in that if one subtree has no solutions then the other

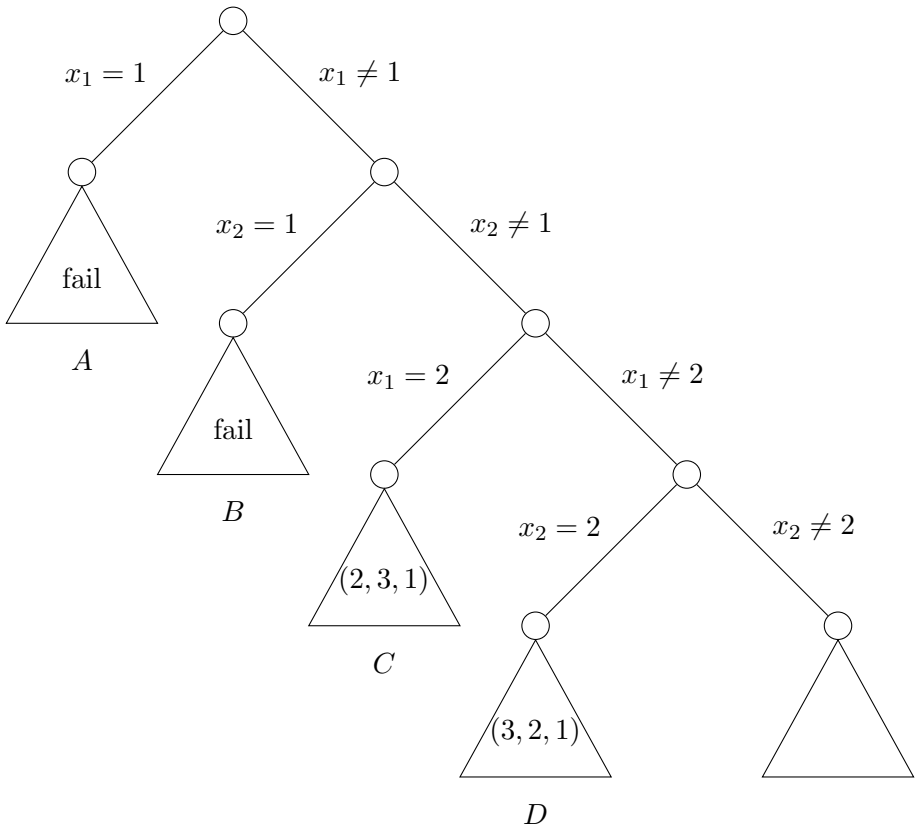


Figure 2.5: Partial search tree with equivalent subtrees due to symmetry. Subtrees A and B are equivalent: neither has any solutions. Subtrees C and D are equivalent: a solution in C (e.g. $\{x_1 = 2, x_2 = 3, x_3 = 1\}$) has a symmetric counterpart in D (e.g. $\{x_1 = 3, x_2 = 2, x_3 = 1\}$).

subtree also has no solutions, and every solution present in one subtree has a symmetric counterpart in the other subtree. As a result, if two subtrees are identified as symmetrically equivalent, only one needs to be explored.

Example 9. Recall the CSP from Example 8 $P = (X, D, C)$ where $X = \{x_1, x_2, x_3\}$, $D = \{1, 2, 3\}$ and $C = (x_1 + x_2 = 5) \wedge (x_3 < 3)$. Part of a possible search tree for this problem is shown in Figure 2.5. □

We now introduce some common, orthogonal classes of symmetries. A *variable symmetry* is a permutation of the variables that is a constraint/solution symmetry (Puget, 2002). Since the inverse of any such permutation is also a symmetry, we will use $\langle x_1, x_2, \ldots, x_n \rangle \leftrightarrow \langle x_{1'}, x_{2'}, \ldots, x_{n'} \rangle$, where $\{x_1, \ldots, x_n\} = X = \{x_{1'}, \ldots, x_{n'}\}$, to denote the variable symmetry which maps every x_i to $x_{i'}$ (or every $x_{i'}$ to x_i). For simplicity of notation, if we have $\{x_1, \ldots, x_k\}, \{x_{1'}, \ldots, x_{k'}\} \subset X$, then $\langle x_1, \ldots, x_k \rangle \leftrightarrow \langle x_{1'}, \ldots, x_{k'} \rangle$ denotes the symmetry which maps each x_i to $x_{i'}$ leaving the remaining variables unchanged.

A *value symmetry* is a permutation within the sets in D (i.e., a bijection from the values of a variable to values of that variable) that is a constraint/solution symmetry (Puget,

2002). We will use $\langle d_{i1}, d_{i2}, \ldots, d_{in}\rangle \leftrightarrow \langle d_{i1'}, d_{i2'}, \ldots, d_{in'}\rangle$, where $\{d_{i1}, d_{i2}, \ldots, d_{in}\} = D_i = \{d_{i1'}, d_{i2'}, \ldots, d_{in'}\}$, to denote a value symmetry for a given variable $x_i \in X$. A *variable-value* symmetry is a permutation of the literals (i.e. the set $V \times D$) that is a constraint/solution symmetry, and is not a variable symmetry or a value symmetry. Note that a variable-value symmetry of a CSP is not necessarily a composition of a variable symmetry and a value symmetry of that CSP (i.e., one or both might not be symmetries of that CSP). These kind of symmetries will be referred to as *non-compositional* variable-value symmetries.

Example 10. The 8-queens CSP in Example 1 has 8 symmetries, including the identity. These symmetries include:

- the variable symmetry $\langle x_1, x_2, x_3, x_4\rangle \leftrightarrow \langle x_8, x_7, x_6, x_5\rangle$, representing the reflection around a horizontal axis through the centre of the board,
- the value symmetry $\langle 1, 2, 3, 4\rangle \leftrightarrow \langle 8, 7, 6, 5\rangle$, representing a similar, vertical symmetry,
- the variable-value symmetry that maps every $x_i = j$ to $x_j = i$, representing a reflection around the top-left/bottom-right diagonal.

These three symmetries are shown in Figure 2.6. Note that the third symmetry cannot be obtained by composing the variable and value symmetries of the problem. The remaining symmetries of the problem can be obtained from those given here by composition. For example, the 90 degree clockwise rotation symmetry can be achieved by first applying the diagonal reflection symmetry and then the reflection around the vertical axis. □

The symmetries of an object form a *group*. A group is a set G and an operator $* : G \times G \to G$ that together satisfy the following laws:

- the operator is associative: $(a * b) * c = a * (b * c)$
- there is an identity element: $\exists e \in G. \forall a \in G. a * e = e * a = a$
- each element has an inverse: $\forall a \in G. \exists b \in G. a * b = b * a = e$

Where the definition of the operator $*$ is clear or is unnecessary, we abbreviate $\langle G, *\rangle$ to G. The *order* of a group is the cardinality of the set G. The symmetries of an object then form a group where the set G is the set of symmetries, and the operator is function composition.

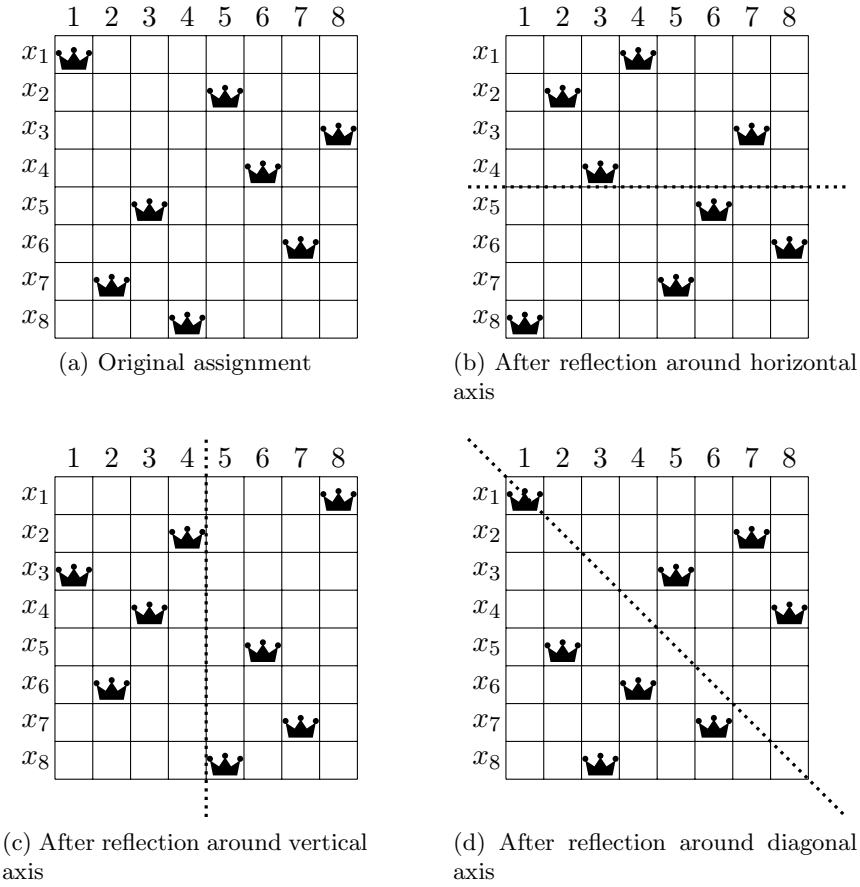
Example 11. The symmetries of the 8-queens problem form a group of order 8. The elements of the group are:

a the identity

b reflection around the central horizontal axis

c reflection around the central vertical axis

d reflection around the top-left/bottom-right diagonal



(a) Original assignment

(b) After reflection around horizontal axis

(c) After reflection around vertical axis

(d) After reflection around diagonal axis

Figure 2.6: Symmetries of an assignment for the 8-queens problem.

e reflection around the top-right/bottom-left diagonal

f 90 degree clockwise rotation

g 180 degree clockwise rotation

h 270 degree clockwise rotation

□

A group $\langle G, * \rangle$ can be *generated* from a set X if every element in G can be obtained by a finite sequence of applying $*$ to elements of X and their inverses. Such a set X is called a *generating set* of G. Importantly, there may be many distinct generating sets for a group. For a given generating set, an element of that set is called a *generator*.

Example 12. One generating set of the symmetry group of the 8-queens problem is $\{b, f\}$ where the labels are as in Example 11. All other elements can be written as the products of these two elements and their inverses. For example $a = b * b^{-1}$, $c = f * f * b$, etc. This is not the only generating set; for example, $\{c, f\}$ is also a generating set. □

Software packages, such as GAP (Group, 2006), are able to represent and manipulate groups efficiently. They are able to compute properties of groups such as the intersection of groups, the orbits of group elements and membership of elements in groups.

2.4 Graphs

A *graph* is a pair $\langle V, E \rangle$ where V is a set of vertices and E is a set of edges (u, v) such that $u, v \in V$. For our purposes we assume that the edges are undirected (i.e. for every edge $(u, v) = (v, u)$) and that each edge has distinct endpoints (i.e. for every edge (u, v), $u \neq v$).

An *automorphism* f of a graph $\langle V, E \rangle$ is a permutation of the vertices such that $\forall (u, v) \in E : (f(u), f(v)) \in E$. That is, an automorphism permutes the vertices of the graph in such a way that the graph remains unchanged.

A graph may be *coloured*. A coloured graph is a triple $\langle V, E, c \rangle$ where V and E are as before, and c is a function $V \to C$, where C is a set of colours. An automorphism of a coloured graph is as for an uncoloured graph, with the additional restriction that each vertex must be mapped to a vertex of the same colour. Formally, an automorphism f of a coloured graph $\langle V, E, c \rangle$ is a permutation of the vertices such that $\forall (u, v) \in E : (f(u), f(v)) \in E$ and $\forall v \in V, c(f(v)) = c(v)$.

Example 13. The graph shown in Figure 2.7a can be reflected across its vertical axis resulting in that of Figure 2.7b, where the dashed arrows indicate the vertex permutation used for this reflection. Since the graph edges are preserved, the permutation is an automorphism. Consider now the graph shown in Figure 2.7c where colours are represented by shading patterns. Its reflection over the horizontal axis results in Figure 2.7d, where the dashed arrows again indicate the associated vertex permutation. This permutation is also an automorphism. Note that reflecting the graph across the vertical axis no longer results in an automorphism due to the vertex colours. □

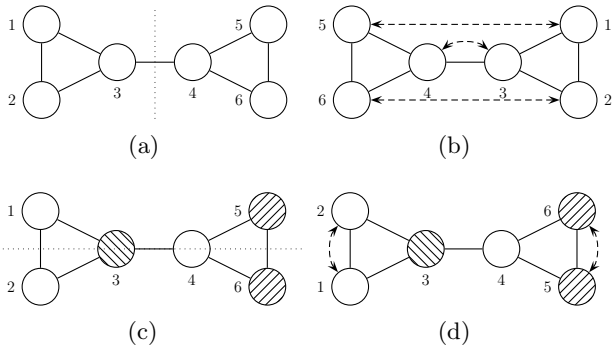


Figure 2.7: Two graphs and one of their possible automorphisms.

Chapter 3

Symmetry Detection

3.1 Introduction

The purpose of studying symmetries in constraint programming is to use the symmetries that are present in a problem in order to solve the problem more quickly, or to eliminate redundant solutions. In order to exploit a problem's symmetries, it is necessary to know what symmetries exist in a problem, either by finding them before the search begins or determining them while the search is in progress. In this chapter we discuss the task of automatic *symmetry detection*, which is to determine automatically what symmetries (if any) a given problem has.

First, we discuss the work that has previously been done in the area of automatic symmetry detection and give a brief chronological survey of existing definitions and methods. Symmetries may be detected automatically in several different ways. One may find all of the solutions to a problem and then examine the set of solutions to determine the symmetries of the problem. Of course, this approach is hindered by the requirement of finding all of the solutions; for many problems it would take an impractical amount of time to do so. Another method is to examine the constraint problem expressed by its concrete specification in some modelling language: if the problem involves two variables x and y and those two variables can be interchanged while leaving the program unchanged, then x and y are symmetric. Such an approach would find some, but not all, of the symmetries of the problem because some symmetries appear only when one considers the meaning of the constraints, rather than their concrete expression.

In this chapter we focus on another approach to symmetry detection that is to use a graph-based representation of a constraint problem. In particular, we discuss in some detail the work presented by (Puget, 2005a) who demonstrated a method of automatic symmetry detection that gives good practical results. This approach relies on work in graph theory that allows us to find symmetries in graphs in a reasonable amount of time. If we are able to transform a constraint problem into a graph, we can then find the symmetries of that graph which correspond to symmetries of the problem. It is necessary that the transformation of a problem into a graph is done in such a way that the resulting graph is not too large, or the time and space needed for the transformation may become prohibitive.

We present a new method of constructing graphs for symmetry detection, similar in spirit to Puget's, and prove that the construction gives correct results. We also describe how we have implemented a system that takes a constraint program written for the ECL^{i}PS^e system, automatically finds its symmetries using our method and can then solve the problem using automatic symmetry breaking to speed up the search. Parts of this chapter have previously been published in (Mears et al., 2009).

3.2 Background

The field of symmetry analysis in constraint programming has been explored for many years. However, the terminology used and even the fundamental definitions of symmetry differ considerably among the work. In this section we describe some of the important kinds of symmetry that have been subject to study.

A symmetry in a problem is, in a sense, a redundancy in the problem: a symmetry indicates that parts of the problem can be removed without affecting the solutions to the problem. Freuder (1991) developed the notion of *interchangeability* in constraint problems as one such form of redundancy. He proposed that if two values are interchangeable, then it is necessary to keep only one of them and discard the other. Freuder provided the definition of a general form of interchangeability, given below.

Definition 1. (Freuder, 1991) Two values a and b for a variable x are *fully interchangeable* if and only if:

1. every solution that assigns b to x remains a solution when c is substituted for b, and

2. every solution that assigns c to x remains a solution when b is substituted for c.

In general, the effort required to find all fully interchangeable values is equivalent to that needed to find all solutions to the problem. Freuder provided algorithms to detect this general form of interchangeability, and also methods to find certain restricted forms of interchangeability that are faster to compute because these kinds of interchangeability can be detected by examining the problem without having to find all of its solutions.

As discussed in the previous chapter, there is a distinction between those symmetries that can be found by examining the constraints of a CSP and those that require all of the solutions to be found and inspected. Following the terminology of Cohen et al. (2005), we call the former *constraint symmetry* and the latter *solution symmetry*. Benhamou (1994) also made this distinction in the context of value interchangeability, calling the former *syntactic* symmetry and the latter *semantic* symmetry.

Other definitions of symmetry focus on the variables of the problem. Puget (1993) describes the concept of a symmetric constraint as a natural counterpart of a mathematical symmetric relation. Such a constraint may be a simple binary constraint like $x = y =$ or $x \neq y$, where the operator is commutative and therefore the variables x and y are symmetric, or a more complex one such as the *count* constraint that is introduced in the same work. The count constraint acts on a set of variables V, a value d and an integer variable c, and ensures that exactly c variables among V take the value d. The variables in V are treated symmetrically according to the definition of the constraint.

Roy and Pachet (1998) present a similar form of symmetry, where two variables are *intensionally permutable* if they have the same domain and any constraints involving either variable must involve both and act on each variable in the same way. Determining which variables are intensionally permutable is done by partitioning the variables into *intensional permutability classes* (or IP-classes). Variables that are in the same IP-class will be treated the same way during problem resolution, that is, their domains will be pruned in the same way. Each constraint separates the variables it acts upon into IP-classes, and the entire problem's IP-classes are computed by combining the IP-classes of each constraint. For example, the all-different constraint treats all of its variables identically and, therefore, all of its variables are in one IP-class. The count constraint described above would have the variables V in one IP-class and the variable c in a singleton IP-class.

The definitions mentioned above define symmetries that act only on values, or only on variables. Some symmetries are not an instance of either of these kinds.

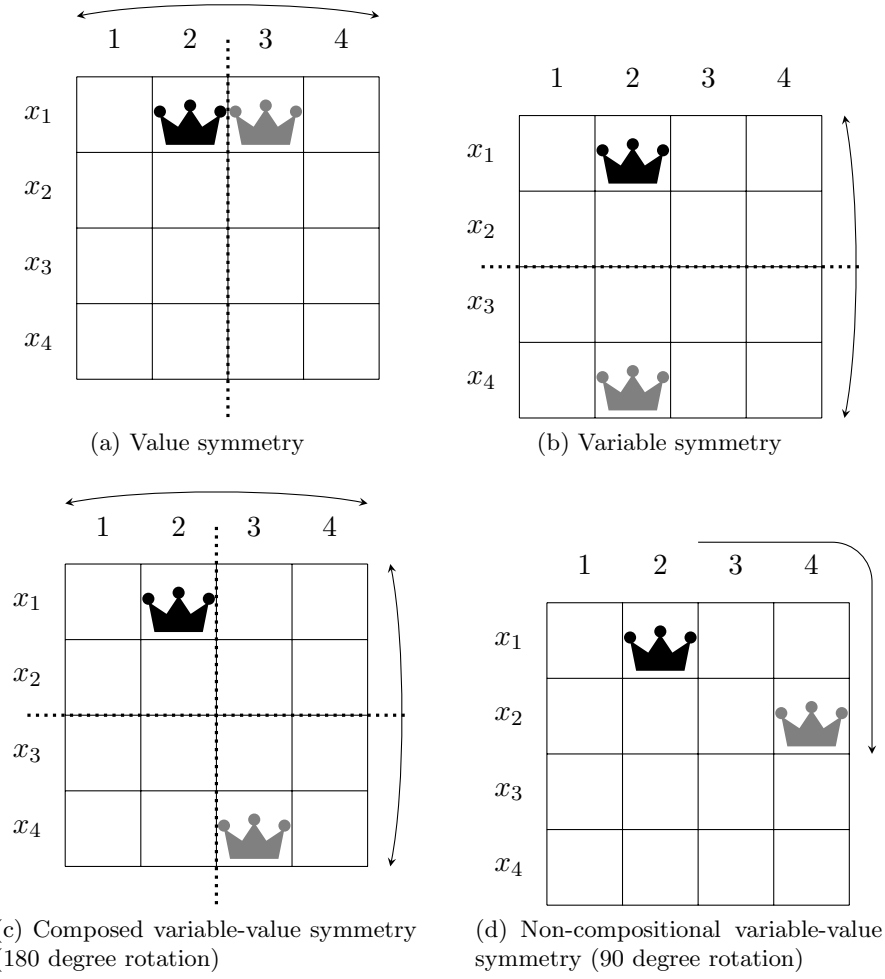
Example 14. For example, in the integer model of the N-queens problem (see Section A.1.1), a 180 degree rotation of the board involves both the variables and the values. Figure 3.1 shows some of the symmetries of the problem. The horizontal reflection of the board corresponds to a variable symmetry (Figure 3.1a) and the vertical reflection to a value symmetry (Figure 3.1b). Combining these two gives another symmetry that involves both the variables and the values (Figure 3.1c). The fourth symmetry (Figure 3.1d) involves the variables and the values, but cannot be obtained by composing any variable symmetries and value symmetries of the problem. □

Meseguer and Torras (2001) give a broader definition that allows a symmetry to be the composition of a variable and value symmetry, which admits the example of the 180 degree chess board rotation. Their definition of a permutation is the composition of a permutation of the variables θ and a permutation of the values σ, so that a literal $x_i = j$ is mapped to $x_{\theta(i)} = \sigma(j)$. In the N-queens example of Figure 3.1c, $\theta(i) = \sigma(i) = n - i + 1$ and maps the queen $x_2 = 1$ to $x_3 = 4$. However, there are still symmetries not covered by this definition, such as the 90 degree rotation of a chess board (Figure 3.1d).

Cohen et al. (2005) give a general definition of symmetry that captures any form of symmetry. This was the solution symmetry introduced, in the previous chapter, as a permutation off the literals of a problem that leaves the set of solutions unchanged. Similarly they define a constraint symmetry as a permutation of the literals that leaves the constraints unchanged.

Regarding symmetry detection, the Cgrass (Frisch et al., 2003b) system for transforming constraint problems performs some detection of constraint symmetry. It detects interchangeable variables by testing pairs of variables, using the transitivity of interchangeability to minimise the number of tests. A pair of variables is deemed to be interchangeable if the set of constraints remains the same after the two variables are exchanged wherever they appear in the set. Determining whether the two constraint sets are the same involves a comparison of the normalised versions of the sets, so that equivalence is not solely syntactic. For example, with the constraint set $\{a < b + 1,\ 1 + b > c\}$ it can be seen that a and c are symmetric, although a naive syntactic comparison would not show it.

Cgrass can also detect symmetries among terms that are larger than simple variables. The search for this kind of symmetry is guided by *structural equivalence*. Two terms are



(a) Value symmetry

(b) Variable symmetry

(c) Composed variable-value symmetry (180 degree rotation)

(d) Non-compositional variable-value symmetry (90 degree rotation)

Figure 3.1: Variable, value, and variable-value symmetries. The grey queen is the symmetric image of the black queen.

structurally equivalent if they are identical once every variable is replaced by a single symbol. For example, the terms $\frac{a*b}{c}$ and $\frac{d*e}{f}$ are structurally equivalent because they have the same structure $\frac{X*X}{X}$.

To determine if this structural equivalence is actually a symmetry of the problem, the system exchanges throughout the constraint set simultaneously each pair $(a, d), (b, e), (c, f)$ and tests the resulting constraint set for equivalence.

Since it is impractical to detect all solution symmetries for anything but the smallest of problems, for the remainder of the chapter we focus on constraint symmetries. The set of constraint symmetries of a problem is a subset of the set of solution symmetries, because every constraint symmetry is also a solution symmetry. The most general form of constraint symmetry, which allows a symmetry to be any permutation of the literals of the problem, can be found by converting a CSP into a graph and then finding the automorphisms of that graph. These automorphisms then correspond directly to symmetries of the problem.

In general, any CSP can be converted into a graph, called the *microstructure* (Jégou, 1993). The microstructure of a problem is a graph where there is a node for each literal of the problem, and an edge between any pair of literals which is explicitly allowed by a constraint, or allowed because there is no constraint between the literals' variables. Also useful is the notion of the *microstructure complement* graph, which has the same nodes as the microstructure but an edge for each pair of literals which are incompatible due to a constraint, or due to their being different literals of the same variable.

Depending on the constraints of a problem, the microstructure (or microstructure complement) graph can contain so many edges that it is impossible to use in practice. Puget (2005a) presents a method to find constraint symmetries via a more compact coloured graph that achieves good results for several problems. Puget's method, and our extensions to it, are presented in the remainder of the chapter.

3.2.1 Puget's coloured graphs

The symmetry detection method presented by Puget (2005a) has two steps. The first takes a CSP $P = (X, D, C)$ and constructs a coloured graph that represents it. Recall that, as introduced in Section 2.4, a coloured graph is a graph represented by the triple $\langle V, E, c\rangle$ where V is a set of nodes, E is a set of edges and c is a map from V onto a set of colours. The purpose of colouring the nodes is to give them a "meaning" and ensure that only nodes of the same kind are interchanged. The second step finds the automorphisms of this coloured graph.

Puget presents two possible forms of graph construction. The first, which we will call the *variable graph* construction, has a node for each variable in X and a node for each constraint in C. Variable nodes have the same colour, and two constraint nodes have the same colour if they represent the same kind of constraint (e.g. $x_1 < x_2$ and $x_3 < x_4$ have the same colour). How constraint and variable nodes are linked depends on whether the constraint is described intensionally or extensionally. For an intensional constraint $c \in C$, an edge is added between the constraint node representing c and the node of each variable in $vars(c)$. Dummy nodes might be required to prevent the appearance of symmetries that do not truly occur in the constraint. Automorphisms of this graph correspond to variable

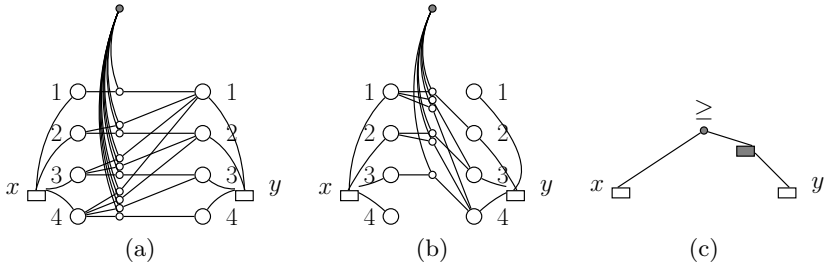


Figure 3.2: Variable graphs using allowed extensional, disallowed extensional, and intensional constraints.

symmetries and thus a variable graph that uses intensional constraints can detect only variable symmetries.

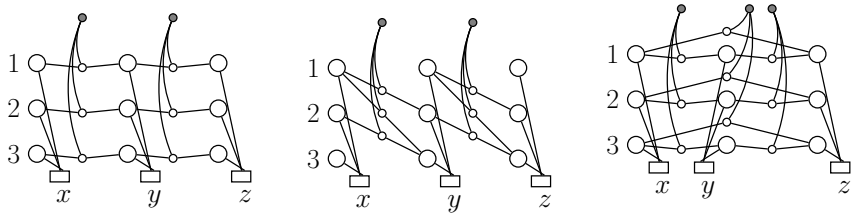
For an extensional constraint $c \in C$, a *value* node is introduced for each value of each variable in $vars(c)$, and an *assignment* node is created for each assignment allowed by c. Edges connect each value node to its variable node, each assignment node to the value node representing each literal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

Example 15. Consider the CSP $(\{x, y\}, \{1, 2, 3, 4\}, \{x \geq y\})$. Figures 3.2a and 3.2b show the variable graphs obtained by using the extensional constraint representation with allowed and disallowed assignments, respectively. Both graphs have 1 constraint node (shown as grey), 2 variable nodes of the same colour (shown as square), 8 value nodes of the same colour (shown as large and white), and either 10 or 6 assignment nodes, respectively, of the same colour (shown as small and white). Figure 3.2c shows the variable graph obtained for the same CSP using the intensional constraint representation. This graph needs 1 constraint node and 2 variable nodes as before, plus a dummy node (shown as a dark square) to prevent the appearance of symmetry.

□

Example 16. Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x = y, y = z\})$. Figure 3.3a shows the graph obtained by using the extensional constraint representation with allowed assignments. The graph has three value nodes per variable.[1] It also has three assignment nodes corresponding to each allowed assignment $\{x = 1, y = 1\}, \{x = 2, y = 2\}, \{x =$

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



(a) $\{x = y, y = z\}$ (b) $\{x < y, y < z\}$ (c) $\{x = y, y = z, x = z\}$

Figure 3.3: Graphs for CSP=($\{x, y, z\}$,$\{1, 2, 3\}$,C) where C is given in the caption of each sub-figure.

$3, y = 3\}$ of $x = y$, and another three for those of $y = z$. All assignment nodes have the same colour (small and white). Finally, the graph has two constraint nodes, each connected to its associated assignment nodes and mapped to the same colour (grey), since they represent constraints of the same kind (equality). The graph associated with CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y, y < z\})$ using the extensional constraint representation with allowed assignments is shown in Figure 3.3b. □

While the method described above is correct (any automorphism of the graph corresponds to a symmetry of the CSP), it is not complete (some CSP symmetries might not appear in the graph). This was demonstrated by Puget (2005a) using the graph of Figure 3.3a which represents constraints $\{x = y, y = z\}$. The graph does not contain any variable symmetries involving y, even though both $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$ and $\langle y, z\rangle \leftrightarrow \langle z, y\rangle$ are symmetries of the CSP. To reduce this problem, Puget suggests to take the transitive closure of equality and less-than constraints, and also to replace pairs of constraints such as $x \leq y$ and $y \leq x$ by $x = y$. Applying this to the graph of Figure 3.3a leads to the addition of $x = z$, and results in the graph of Figure 3.3c, which does contain symmetries $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$ and $\langle y, z\rangle \leftrightarrow \langle z, y\rangle$. Note, however, that the transitive closure does not make the method complete.

Furthermore, and as mentioned before, there are symmetries of the CSP that cannot be expressed as the composition of variable and value symmetries present in the CSP. These non-compositional symmetries cannot be expressed using the intensional representation of constraints proposed by Puget since, as he indicates, it is only suitable for variable symmetries. Neither can they be represented using the extensional representation, due to the existence of variable nodes and to the different colour used for different kinds of constraints.

Puget addresses this issue with a second form of graph construction, which we will call the *boolean graph* construction. This construction is based on the notion that any CSP with finite domain variables can be converted into an equivalent CSP with boolean variables. The direct encoding (Walsh, 2000) represents the problem $P = (X, D, C)$ with one boolean variable x_{ij} for each literal $x_i = j$ of P, where $x_i \in X$ and $j \in D_i$. The variable x_{ij} takes the value true if and only if x_i should take the value j. An allowed assignment of a constraint in P corresponds to an assignment of *true* to all the boolean variables

representing literals in the assignment. A solution to P corresponds to an instantiation of the boolean variables such that:

- precisely one boolean variable for each variable $x_i \in X$ is set to *true*, and
- each constraint in P has at least one allowed assignment.

Puget proposes a graphical representation of the original CSP based on this boolean model. Each literal $x_i = j$ of the problem is represented by a *literal node* x_{ij} (note that there are no variable nodes). Each primitive constraint, such as $x = y$, is represented by a constraint node and one assignment node per assignment allowed by the constraint. The constraint node is linked to all assignment nodes for the constraint, and each assignment node is linked to the literal nodes involved in that assignment. A CSP with only one primitive constraint is therefore represented by a graph that is very similar to the extensional form of the variable graph construction.

Example 17. Consider the CSP $P = (\{x, y\}, \{1, 2, 3\}, \{x = y\})$. Figure 3.4 compares the variable graph construction with extensional constraints and boolean graph construction. In the boolean graph construction, there are no variable nodes and therefore some non-compositional variable-value symmetries can be represented.

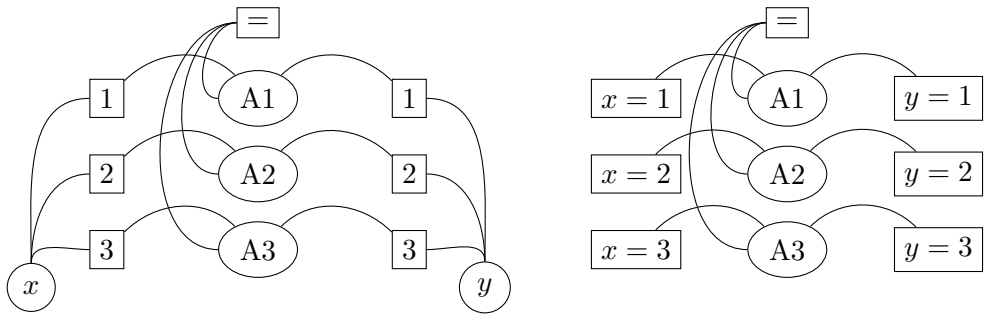


Figure 3.4: Extensional constraint in variable graph construction (left) compared with boolean graph construction (right).

□

While the boolean graph representation has significant advantages over the variable graph representation, it also has two significant problems. First, as is, Puget's definition of the boolean graph is incorrect. In particular, automorphisms of this graph representation do not necessarily correspond to solution symmetries of the original CSP. A counter example can be illustrated with the CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y\})$ (shown in Figure 3.10 in Section 3.4.3). The permutation $\langle x = 3\rangle \leftrightarrow \langle z = c\rangle$, for any $c \in \{1, 2, 3\}$, is an automorphism of the graph but is not a symmetry of the CSP. In Lemma 2 in Section 3.4.4, we impose some restrictions on the boolean representation to guarantee that graph automorphisms do indeed correspond to symmetries.

Second, the link to the original constraint node precludes symmetries involving different kinds of constraints (such as an all-different and a disequation). Interestingly, Puget's boolean model for the all-different constraint (in (Puget, 2005a) and explained below) does

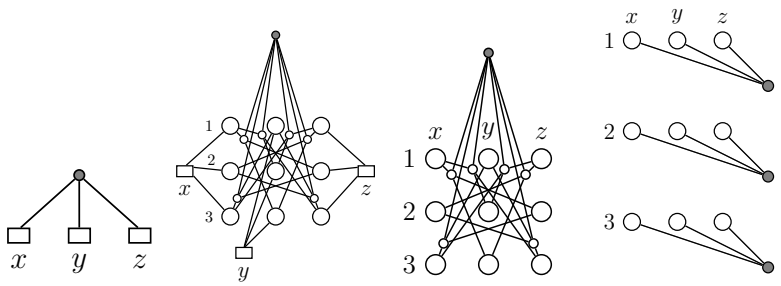


Figure 3.5: Representations of CSP $(\{x, y, z\}, \{1, 2, 3\}, \{all\_different(\{x, y, z\})\})$

not appear to include a link back to the original constraint node. For clarity, we briefly summarise four alternative representations for this constraint under Puget's different constructions. Note that these are our interpretations of Puget's constructions, as the original work (Puget, 2005a) does not provide a formal definition.

The first and second representations use the variable graph construction while the third and fourth representations use the boolean graph construction. The first representation is an intensional representation, consisting of a node for the constraint itself, a node for each variable in its scope, and an edge between the constraint node and each variable node (Figure 3.5a). The second representation is the extensional representation with $O(m^n)$ nodes representing allowed assignments of n variables each with a domain size of m (Figure 3.5b). The third representation is the standard boolean one with an assignment node for each allowed assignment. Note again that this is almost the same as the extensional representation, but without any nodes corresponding to the original variables (Figure 3.5c). The fourth representation also uses the boolean graph construction, but is a specialised form devised particularly for the all-different constraint. In this representation, the constraint is reformulated as a set of special constraints over the boolean variables; each constraint states that only one of the connected boolean variables is true (Figure 3.5d, where each black circle is one of these new constraints). The all-different constraint over n variables v_i can be represented by m constraints (one per value, c_j), each of which is connected to the n boolean variables $b_{ij} : i \in 1..n$, representing $v_i = c_j$.

It can be seen from Figure 3.5 that the graph using the special representation of all-different (Figure 3.5d is much smaller ($O(m.n)$ nodes) than the graph that uses the extensional form (Figure 3.5c) of the all-different constraint ($O(m^n)$ nodes). The automorphisms of these two graphs represent the same symmetries.

Using a special representation for a global constraint such as all-different, rather than decomposing it into smaller constraints, can reduce the amount of symmetry found.

Example 18. Consider the CSP $(\{x_1, x_2, x_3, x_4\}, \{1, \ldots, N\}, \{all\_different(X), x_2 \neq x_4\})$, where all-different is represented using N disallowed assignments nodes. Figure 3.6 shows two graphs representing this CSP for $N = 1$. Figure 3.6a shows the boolean graph with the all-different constraint represented in its usual k-ary form and Figure 3.6b show the graph where the all-different constraint is decomposed into not-equal constraints. In the

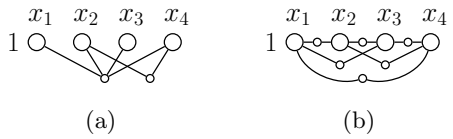


Figure 3.6: Representing all-different using n-ary and binary constraints.

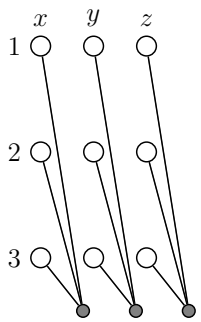


Figure 3.7: Implied boolean constraint.

first graph, the representation of $x_2 \neq x_4$ limits the symmetries of the graph to those generated by $\langle x_1, x_3\rangle \leftrightarrow \langle x_3, x_1\rangle$ and $\langle x_2, x_4\rangle \leftrightarrow \langle x_4, x_2\rangle$. As a consequence, symmetries such as $\langle x_1, x_2\rangle \leftrightarrow \langle x_2, x_1\rangle$ are lost since they do not correspond to automorphisms of the graph. However, in the second graph where all-different is represented by the disallowed assignments of the equivalent constraint $\{x_i \neq x_j | 1 \leq i < j \leq 4\}$, all variable symmetries are present in the graph. □

To complement the special representation of all-different, Puget also proposes a similar representation of the implied constraint between the different values of a single variable, connecting the boolean nodes $b_{ij} : j \in 1..m$ (see Figure 3.7 for an example). This is necessary to ensure graph automorphisms correspond to solution symmetries. We show in the next section that these are not only necessary, but also sufficient for correctness. However, if this is expressed via an all-different constraint, then opportunities for detecting symmetries may be lost since this representation does not allow a disequation between two values to participate in a symmetry with a disequation between two variables (a similar situation is shown in Example 18).

3.2.2 Puget's representation for expressions

Puget's method is based on constraints whose arguments are distinct variables. In order to be able to handle constraints involving expressions, Puget proposes to represent any expression of the form $x_i\ op\ x_j$, where $x_i, x_j \in X$ are distinct variables, as the extensional constraint associated with $op(x_i, x_j, t)$, where t is a new (temporary) variable which is then used to replace the expression $x_i\ op\ x_j$ as the constraint's argument.

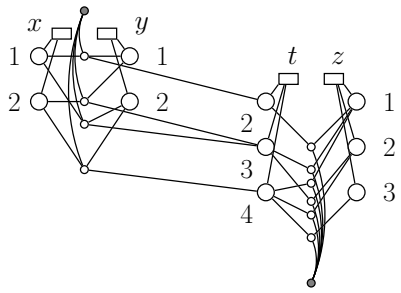


Figure 3.8: Variable graph of a CSP with constraint $x + y > z$. The extra variable t represents $x + y$.

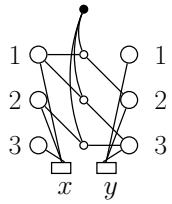


Figure 3.9: Variable graph of a CSP with constraint $x + 1 > y$. The expression $x + 1$ is represented without additional variables.

Example 19. Consider the CSP $P = (\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1, 2, 3\}\}, \{x + y > z\})$, whose variable graph with extensional constraints is shown in Figure 3.8. The expression $x + y$ is represented by a new variable t with domain $D_t = \{2, 3, 4\}$. □

We believe this method was suggested because (a) it reuses the already defined constraint representation, and (b) uses the same colour for constraints with and without complex expressions. For example, $A < B$ can be represented by the same colour as $C < D + 1$ if the latter constraint is expressed as a combination of $E = D + 1$ and $C < E$. This reduces the syntax dependency of the graph and thus might result in more symmetries being detected.

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

Example 20. Consider the CSP $P = (\{x, y\}, \{1, 2, 3\}, \{x+1 > y\})$, whose variable graph is shown in Figure 3.9. The expression $x+1$ is represented without any additional variables. The node $x = 1$ represents $(x + 1) = 2$, and node $x = 2$ represents $(x + 2) = 3$. □

3.3 Problems with Puget's Representation

The graph representations described in the previous section have some flaws that lead to incorrect or suboptimal results. In this section, we describe these problems.

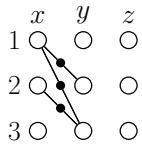


Figure 3.10: Boolean graph construction with automorphisms that are not symmetries of the CSP.

3.3.1 Problems with the Boolean Graph

In some cases the boolean graph construction can give incorrect results. That is, the graph can have automorphisms that correspond to permutations of the CSP's literals that are *not* symmetries of the CSP.

Example 21. Consider a CSP $P = (\{x, y, z\}, \{1, 2, 3\}, \{x < y\})$. Figure 3.10 shows the boolean graph construction for P using allowed assignments for the constraint. The graph has automorphisms that are not solution symmetries; for instance, the automorphism $\langle x = 3\rangle \leftrightarrow \langle z = 3\rangle$ is not a symmetry of P. □

3.3.2 Problems with Expression Representation

As described in Section 3.2.2, expressions involving more than one variable are broken into sub-expressions, each of which is represented by a new (temporary) variable t_i. Constraints involving expressions as arguments are simply treated by replacing each such argument by a new variable representing the expression. As noted by Puget and others (Puget, 2005a; Ramani and Markov, 2004), this can lead to the unintentional loss of symmetries due, for example, to the associative nature of operators.

Example 22. Consider the constraint $x + y + z > w$. If the constraint is parsed as $(x + y) + z > w$, it would be transformed into constraint $t_1 > w$, where the new variable t_1 has associated constraint $+(t_2, z, t_1)$ (representing the constraint $t_2 + z = t_1$), and new variable t_2 has associated constraint $+(x, y, t_2)$. Although in the expression all three variables are interchangeable, the associated graph only has the variable symmetry $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$. □

The problem can be ameliorated (Puget, 2005a; Ramani and Markov, 2004) by representing multiple occurrences of a binary associative operator with a single n-ary operator. For instance, in the previous example we would only introduce one extra variable t_3, and the constraint $+(x, y, z, t_3)$. As recommended by Puget (2005a), asymmetric binary arithmetic operations, such as $x - y$ and x/y, are decomposed using their unary inverse operators, resulting in $x + (-y)$ and $x * (1/y)$. This allows further grouping of associative operators while at the same time preventing the creation of false symmetries. Unfortunately, as mentioned before, this preprocessing only reduces the problem instead of eliminating it, since the intermediate variables can still prevent some symmetries from being captured by the graph, even after performing all the preprocessing steps indicated above.

Example 23. Consider the CSP $(\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1\}\}, \{x + z \neq y, y + z \neq x\})$. This CSP has a variable symmetry $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$, and a value symmetry on variables x

and y that exchanges $\langle 1,2\rangle \leftrightarrow \langle 2,1\rangle$. The expressions in the two constraints are represented by $t_1 = x + z$ and $t_2 = y + z$. The associated extensional graph is shown in Figure 3.11, with grey and black constraint nodes linking assignment nodes for equality and disequality constraints, respectively. It can be seen that the graph captures the variable symmetry (achieved by reflecting the graph in a vertical axis positioned over value node 1 of z), but not the value symmetry (which would be achieved by reflecting the graph in a horizontal axis positioned in between value nodes 1 and 2 of x and y). □

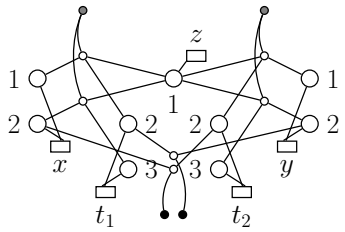


Figure 3.11: Graph of CSP $(\{x, y, z\}, \{\{1,2\},\{1,2\},\{1\}\}, \{x + z \neq y, y + z \neq x\})$

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

3.4 A new graph representation

Puget (2005a) presents experimental results that show that his graph constructions are able to detect many symmetries of benchmark problems with reasonable running time. However, as illustrated in the previous section, the constructions as described do not satisfactorily cover all possible kinds of CSP where one may wish to use automatic symmetry detection (such as those involving set variables), depend strongly on syntax, and, as mentioned before, in some cases may lead to incorrect results.

In this section, we formally define new graph representations of CSPs that can be used to detect symmetries. Our main aim is to explore representations that are able to detect as many symmetries as possible in all kinds of CSPs and to ensure their correctness. While we are also interested in minimising the size of the graph as far as possible, this is secondary to ensuring correctness and accuracy. Our overall motivation is to develop a method that can be used as a foundation for a symmetry detection method that will act on problem models (described in Chapter 4), and as such our method presented here does not need to be used with very large problem instances.

3.4.1 Allowed and disallowed assignments

Instead of using intensional constraints or factoring out expressions with temporary variables, we believe it is cleaner and simpler to return to extensional constraints. An important motivation for us is to eliminate different "kinds" of constraints, which are represented by nodes with different colours and stick to just two kinds: constraints represented extensionally by *allowed* and by *disallowed* assignments. When seen in this light, it becomes clear that we can avoid the representation of temporary variables and constants by absorbing expressions into the constraint in which they appear. We also decided to drop variable nodes and value nodes and, instead, use literal nodes. The consequences of this will be discussed in section 3.4.3 below.

Example 24. Consider the CSP $(\{x, y, z\}, D, \{x + y > z\},)$ where $D_x = D_y = \{1, 2\}$ and $D_z = \{1, 2, 3\}$. The ternary constraint $x + y > z$ can simply be represented extensionally by its set of allowed assignments, $\{\{x = 1, y = 1, z = 1\}, \{x = 1, y = 2, z = 1\}, \{x = 1, y = 2, z = 2\}, \{x = 2, y = 1, z = 1\}, \{x = 2, y = 1, z = 2\}, \{x = 2, y = 2, z = 1\}, \{x = 2, y = 2, z = 2\}, \{x = 2, y = 2, z = 3\}\}$, all linked to an additional constraint node, as illustrated by Figure 3.12. □

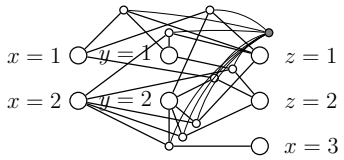


Figure 3.12: Representing expressions as allowed assignments

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

Example 25. Figures 3.13a and 3.13b show the graphs obtained by representing the allowed assignments of the constraints in CSPs (X, D, C) and (X, D, C'), respectively, where $X = \{x, y\}$, $D_x = D_y = \{1, 2\}$, $C = \{x > y, x < y\}$, and $C' = \{x \neq y\}$. It is clear that, without the constraint nodes, the graphical representation of these two CSPs are indistinguishable, even though while the first CSP has no solutions, the second has two. The confusion is due to $x \neq y$ being logically equivalent to $(x > y \vee x < y)$. Figures 3.13c and 3.13d show the graphs obtained by representing the disallowed assignments of the constraints (note that, for clarity, identical assignments disallowed by different constrains

have been merged). As it is clear from the figure, the graphs are perfectly distinguishable even though constraint nodes are not represented. □

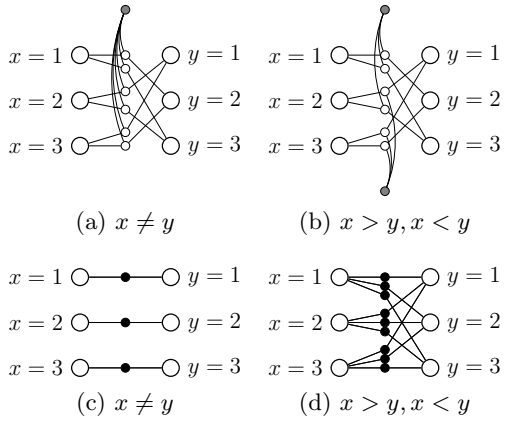


Figure 3.13: Graphs using allowed (a,b) and disallowed (c,d) assignments.

The simplicity of this method is pleasing since it eliminates problems such as the explicit representation of constants (which is now avoided regardless of whether the constant appears as a constraint argument or not), the normalisation required when multiple occurrences of a variable appeared in a constraint (such as A×A=1, which can now be easily treated by computing the values of A for which the constraint is satisfied), the problem of associative and non-symmetric operands appearing in the same constraint, and in general, any such normalisation issue affecting a single constraint. For instance, Figure 3.14 shows the graph obtained for the CSP of Example 23 using the new representation method. The elimination of temporary variables yields a graph that has not only the variable symmetry $\langle x, y\rangle \leftrightarrow \langle y, x\rangle$, but also the value symmetry for variables x and y that exchanges $\langle 1, 2\rangle \leftrightarrow \langle 2, 1\rangle$.

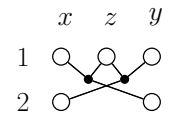


Figure 3.14: Graph for CSP $(\{x, y, z\}, \{\{1, 2\}, \{1, 2\}, \{1\}\}, \{x + z \neq y, y + z \neq x\})$

The method also avoids syntactical issues regarding constraints whose name appears to be different but is actually equivalent (e.g., $x < y$ and $z > w$ should be considered as constraints of the same kind). However, some problems remain when a CSP has two constraints with identical scopes (see section 3.4.3).

3.4.2 Disallowed assignments and the microstructure complement

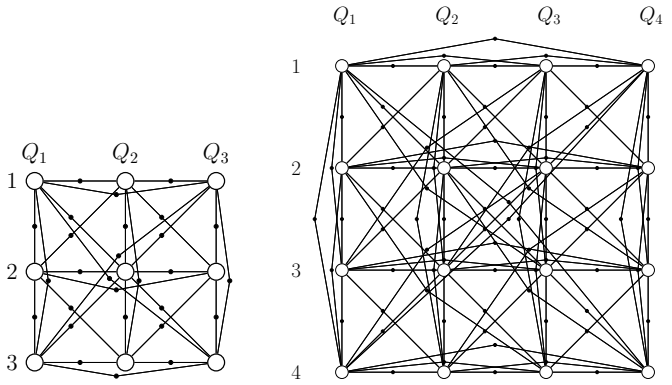
Given the above discussion, it would seem advantageous to represent a CSP using only its disallowed assignments. Let us consider the advantages and disadvantages of such an approach.

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

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

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

Example 26. Figure 3.15 illustrates the disallowed assignments graph for the 3- and 4-queens problems (with disallowed disequality assignments shown as small black nodes). Note the need to represent the disallowed pairs of distinct literals for each variable, to capture all the symmetries of the chessboard. Without these disallowed pairs, the 90 degree rotation symmetry would not be an automorphism of the graph. □



(a) $3-queens$ (b) $4-queens$

Figure 3.15: Graph for instances of N-queens using disallowed tuples.

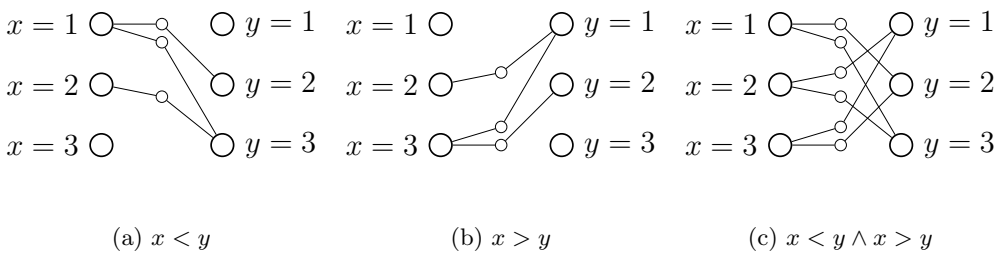
The disallowed assignments graph is similar to the microstructure complement introduced by Jégou (1993) and extended by Cohen et al. (2005) (described in Section 2.3). The only difference between the disallowed assignments graph and the microstructure complement is that each disallowed assignment is represented by an assignment node and the literals linked to it, whilst the microstructure complement represents a disallowed assignment by a single hyperedge linking the literals.

The main drawback of both the microstructure complement and the disallowed assignments graph is their size. Firstly, they have, in general, more nodes than necessary. Every value for every variable is represented as a node, although many of these nodes could never appear in a solution (techniques for pruning nodes will be discussed below). And secondly, the number of hyperedges (or assignment nodes) in the graph is high. A mathematical

equation, such as $x = 2y + z$ requires approximately d^3 hyperedges, where d is the size of the domains of x, y and z. While for some constraints the set of disallowed assignments is the most compact way to represent the constraint, many others, such as mathematical equations, are much more compactly represented by their allowed assignments. Moreover, the microstructure complement requires d^2 edges to disallow multiple assignments for a variable in the CSP with domain size d. Thus, for n variables, nd^2 edges are needed. While this number could be kept to nd using intensional constraints, as described before, this would limit the symmetry possibilities, making non-compositional variable-value symmetries unlikely. Therefore, in the next section we consider an alternative graph in which only allowed assignments are represented explicitly.

3.4.3 Allowed assignments and the microstructure

The microstructure described in Section 3.2 (page 25) suffers from a flaw caused by the same problem that prevented us in Section 3.4.1 from eliminating constraint nodes for allowed assignments: if two or more constraints have the same scope, then the set of hyperedges over that scope represents the disjunction, instead of the conjunction, of the constraints.



(a) $x < y$ (b) $x > y$ (c) $x < y \wedge x > y$

Figure 3.16: Constraints over the same scope.

Example 27. Figure 3.16 shows the result of two constraints over the same scope using allowed assignments. The constraints $x < y$ and $y < x$ (shown in the first two subfigures), when combined, result in the same graph as the constraint $x \neq y$ (shown in the third subfigure). □

This flaw can be easily fixed, however, by a preprocessing step which replaces each set of constraints that have the same scope by a new constraint whose allowed assignments are those that satisfy all constraints. From now on, we will assume that every CSP (X, D, C) has already been preprocessed and, therefore, it is true that for every two distinct constraints $c_1, c_2 \in C : vars(c_1) \neq vars(c_2)$.

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

allowed assignments. Since the number of constraints is typically much smaller than 2^n, the number of hyperedges in the microstructure is typically very large indeed.

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

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

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

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

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

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

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

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

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

assignments over distinct sets of variables and, therefore, they belong to two different constraints. This means that $f(S)$ also satisfies m distinct constraints and, therefore, all constraints in $C \cup BC$. Since $f(S)$ is a complete assignment, it must also be a solution. □

3.4.4 A graph including allowed and disallowed assignments

We now present a new graph representation for CSPs that does not require all constraints to use the allowed (or disallowed) assignments and, thus, permits different constraints to use different assignments. The representation takes many ideas from the work of Puget (Puget, 2005a) but is also closely related to the microstructure and microstructure complement of Cohen et al. (2005). As before, our graph representation of a CSP (X, D, C) has a node for every literal (and, thus, for every value of the domain of every variable in X). However, we now admit both allowed assignments and disallowed ones, distinguished by different colours.

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

Definition 4. *A CSP (X, D, C), where no two constraints have identical scopes, is represented by the* full assignments graph *if the graph contains three kinds of nodes:*

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

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

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

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

Example 28. Consider the CSP $P = (\{x, y, z\}, \{1, 2, 3\}, \{x < y\})$ (the same CSP as in Example 21). Figure 3.17 shows the boolean graph of this CSP as in Figure 3.10, but the now graph has been augmented with disallowed assignments between each pair of literals

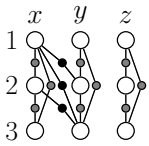


Figure 3.17: Pairs of literals for the same variable linked by disallowed assignments. Black nodes are allowed assignments, grey nodes are disallowed assignments.

in the same variable. The permutation $\langle x = 3\rangle \leftrightarrow \langle z = 3\rangle$ is no longer an automorphism of the graph. □

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

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

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

Let us now show that every constraint in C is satisfied by $f(S)$. If every pair of variables is in the scope of an allowed constraint, then the proof is again as in Lemma 1 above. Otherwise, by the definition of the full assignments graph, there must be a disallowed assignment between every pair of literals with the same variable ($x = a$ and $x = b$, for all $a \neq b$). Consider a pair of literals $lit_1, lit_2 \in S$ and let $v_1 = var(lit_1)$ and $v_2 = var(lit_2)$. Because S is a solution, $v_1 \neq v_2$. For $f(S)$ to be a solution, we must be able to show that $f(lit_1)$ and $f(lit_2)$ are not connected to the same disallowed assignment node. Because S is a solution, lit_1 and lit_2 must not both be connected to any disallowed assignment node. Therefore, $f(lit_1)$ and $f(lit_2)$ cannot both be connected to the same disallowed assignment node.

In addition, if v_1 and v_2 are in the scope of an allowed constraint, we must also show that $f(lit_1)$ and $f(lit_2)$ are connected to an allowed assignment node. Because S is a solution, lit_1 and lit_2 must be connected to some allowed assignment node n. Therefore, $f(lit_1)$ and $f(lit_2)$ are connected to the allowed assignment node $f(n)$.

Since the coexistence of $f(lit_1)$ and $f(lit_2)$ is permitted in a solution, and since $f(S)$ is a complete assignment, $f(S)$ must be a solution. □

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

Proof. It is easy to prove the lemma by contradiction. Consider the CSP represented by $(\{x, y, z\}, \{1, 2, 3\}, \{x < y, y < z\})$ whose only solution is $\{x = 1, y = 2, z = 3\}$. While this CSP has the solution symmetry $\langle x = 2, x = 3\rangle \leftrightarrow \langle x = 3, x = 2\rangle$, this is not a constraint

symmetry since it maps the allowed assignment $\{x = 2, y = 3\}$ of constraint $x < y$ to the disallowed assignment $\{x = 3, y = 3\}$. The full assignment graph for the CSP is shown in Figure 3.18, using disallowed assignments to represent both constraints. It is clear that the solution symmetry is not an automorphism of the graph. □

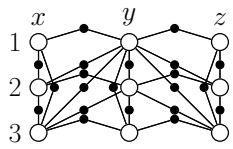
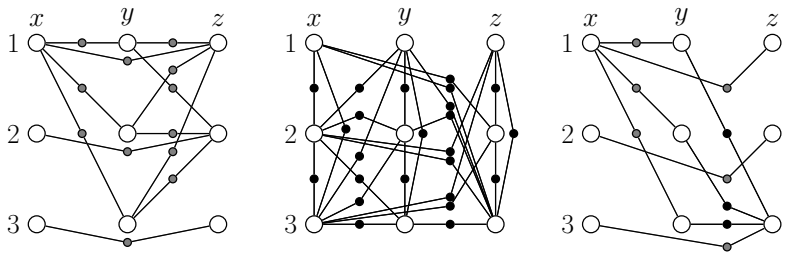


Figure 3.18: Full assignments graph of CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x < y, y < z\})$

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

Example 29. Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, C)$, where $C = \{c_1(z, y), c_2(x, y), x = z\}$, $c_1 = \{1, 2\} \times \{1, 2, 3\}$ and $c_2 = \{1\} \times \{1, 2, 3\}$. This CSP contains a solution symmetry, $\langle x = 1\rangle \leftrightarrow \langle z = 1\rangle$. As illustrated by Figure 3.19, if this CSP is represented using only disallowed assignments or only allowed assignments, the symmetry is present in the graph. However, if we use the representation with the smallest graph (disallowed assignments for $c_1(z, y)$ and allowed assignments for $c_2(x, y)$ and $x = z$), then that symmetry is not present in the graph. □



(a) Allowed graph (b) Disallowed graph (c) Full graph

Figure 3.19: Symmetry $\langle x = 1\rangle \leftrightarrow \langle z = 1\rangle$ is not present in (c) (see Example 29).

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

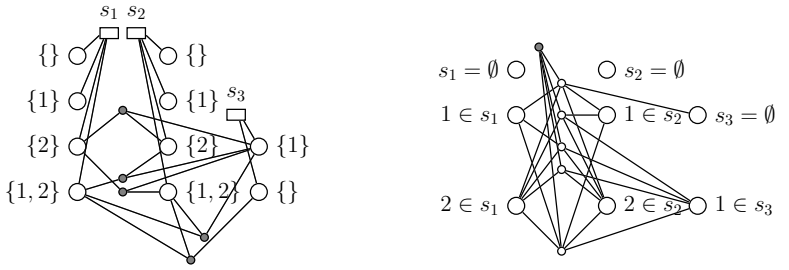
Proof. Immediate since, by definition, any allowed (disallowed) assignments graph is an instance of a full assignments graph in which only allowed (disallowed) assignments are used for representing constraints. □

3.4.5 Representing Sets

While sets are used in modelling many problems – arguably most problems coming from the real world – they are not covered by Puget's method (Puget, 2005a). Consider a set variable $x_i \in X$ known to be a subset of set S. Its domain D_i is equal to the power-set of S, i.e., $D_i = \{S' | S' \subseteq S\}$. In any graph representation which includes value nodes or literal nodes, a node is required for every element of the power set, and this leads to large graphs. Using the extensional method (e.g. in Figure 3.20a) the graph has one variable node and $2^{|S|}$ value nodes (e.g., if $S = \{1, 2, 3, 4\}$ the graph contains 1 variable node for x_i and 16 value nodes representing values $\{\}$, $\{1\}$, $\{2\}$, ..., $\{2, 3, 4\}$, and $\{1, 2, 3, 4\}$). Set constraints are then extensionally represented as usual by using their allowed or disallowed assignments.

An alternative approach is to use a boolean representation, (e.g. in Figure 3.20b). Then, rather than using each element in the powerset of S to create a value node, we would use each element in the set, i.e., we would obtain $|S| + 1$ nodes where one node represents the *empty* set, and $\forall d \in S$ there is a node representing $d_i \in x_i$ (e.g., if $S = \{1, 2, 3, 4\}$ the graph contains 5 nodes representing $x_i = \{\}$, $1 \in x_i$, $2 \in x_i$, $3 \in x_i$, and $4 \in x_i$). A constraint on the set is represented by its allowed assignments. A set's value is represented, in an assignment, by its elements. Therefore, the assignment includes the nodes of all the elements which belong to the set value (or the empty set node if the set value is the empty set). The constraint is, as usual, the *disjunction* of the represented assignments. In the graph, a constraint is represented by a constraint node and a node for each of its allowed assignments. The assignment node is then linked to each of the nodes representing an element in the set value, or the node representing the empty set. This boolean representation results in fewer nodes but more edges.

Example 30. Figure 3.20a and 3.20b show the two alternative graphs obtained for CSP $(\{s_1, s_2, s_3\}, D, \{|(s_1 \cap s_2) \cup s_3| = 2\})$ where $D_{s_1} = D_{s_2} = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$, and $D_{s_3} = \{\{\}, \{1\}\}$.



(a) Extensional (b) Boolean

Figure 3.20: Different representations for set constraint $|(s_1 \cap s_2) \cup s_3| = 2$

□

The significance of these alternative representations for our implementation is shown in section 3.6.1 below.

3.5 Reducing graph size

While the full assignment graph might result in smaller graphs than those obtained using only allowed or only disallowed assignments, even full assignments graphs tend to be rather large (see Table 3.1 for size data). Consider, for example, the number of nodes in the full assignment graph of a CSP, which is the sum of:

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

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

3.5.1 Minimising the number of assignment nodes

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

Example 31. Consider the CSP (X, D, C) where $X = \{x, y, z\}$, $D = \{1, 2, 3\}$ and $C = \{alldifferent(X)\}$. Using allowed assignments, the graph would require the following 6 hyperedges, which are flattened into $6 \times 3 = 18$ edges.

$x=1-y=2-z=3$	$x=1-y=3-z=2$	$x=2-y=1-z=3$
$x=2-y=3-z=1$	$x=3-y=1-z=2$	$x=3-y=2-z=1$

Using disallowed assignments, the graph would require the following 21 hyperedges, which are flattened into $21 \times 3 = 63$ edges.

$x=1-y=1-z=1$	$x=1-y=1-z=2$	$x=1-y=1-z=3$
$x=1-y=2-z=1$	$x=1-y=2-z=2$	$x=1-y=3-z=1$
$x=1-y=3-z=3$	$x=2-y=1-z=1$	$x=2-y=1-z=2$
$x=2-y=2-z=1$	$x=2-y=2-z=2$	$x=2-y=2-z=3$
$x=2-y=3-z=2$	$x=2-y=3-z=3$	$x=3-y=1-z=1$
$x=3-y=1-z=3$	$x=3-y=2-z=2$	$x=3-y=2-z=3$
$x=3-y=3-z=1$	$x=3-y=3-z=2$	$x=3-y=3-z=3$

Finally, breaking the all-different constraint into binary constraints requires the following 9 edges:

$$\begin{array}{lll} x=1-y=1 & y=1-z=1 & x=1-z=1 \\ x=2-y=2 & y=2-z=2 & x=2-z=2 \\ x=3-y=3 & y=3-z=3 & x=3-z=3 \end{array}$$

□

Breaking down each constraint into a logically equivalent conjunction of constraints with as small scope as possible, has a very useful side-effect: it will tend to increase the number of constraints with the same scope, which can be integrated (during the preprocessing step) into a single constraint, thereby increasing the number of detected symmetries (see Example 18).

Note that we can also represent the all-different constraint extensionally using only $O(d.k)$ nodes. Suppose we order the k variables in the constraint, $v_1, \ldots, v_k$. We will use boolean variables b_{jc} to represent $v_j = c$. We now introduce, for each value c, k new boolean variables $t_{jc} : j \in 1..k$. t_{jc} is *true* if any of $b_{1c}, \ldots, b_{jc}$ are *true*. Let $t_{1c} = b_{1c}$, and for each $j \in 2..k$ there is a constraint with scope $t_{j-1,c}, b_{j,c}, t_{j,c}$ defined by three allowed assignments: $< 1, 0, 1 >, < 0, 1, 1 >, < 0, 0, 0 >$. There are just k such constraints for each value c, and each constraint has just three assignments, so the total number of edges required is $3.d.k$ which is $O(d.k)$. While this representation is very compact, it has the same problem as the boolean representation proposed by Puget: it might lead to a loss of symmetry detection if it has to interact with that obtained for other kinds of constraints. Since our main aim is accuracy rather than reducing the graph size, we have not used it in our implementation.

Minimising the number of variables in the scope of constraints has other advantages. If all constraints represented in the graph have less than k variables in their scope, then the number of edges is less than n^k where n is the number of nodes. Moreover, as pointed out in by Cohen et al. (Cohen et al., 2005), by adding enough disallowed assignments over k variables it is possible to create a (microstructure complement) graph whose automorphisms represent all solution symmetries of the CSP. Their proof is easily adapted to show that the same holds true of the full assignment graph.

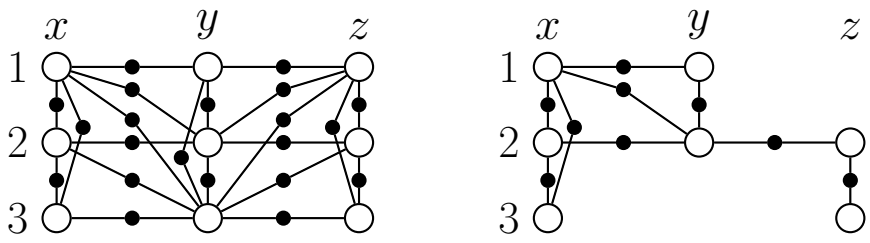
This result gives an upper bound on the size of the graph needed to capture all the solution symmetries of a given CSP. It follows that by representing constraints with an equivalent conjunction of constraints of minimum possible arity, we achieve a lower bound on this worst case. Naturally, it remains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

3.5.2 Minimising the number of literal nodes

CSPs can be simplified by using standard propagation techniques that reduce the domains of the variables and, thus, the number of literal nodes that appear in the full assignment graph. Correct simplifications to achieve node- and arc-consistency (see Section 2.2.1) are

well-known, and yield a reduced CSP that has the same set of solutions as the original one. Indeed, consistency algorithms were first devised for improving the efficiency of picture recognition programs, to reduce the size of the graph, which is exactly what we are seeking to do!

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



(a) Before arc-consistency. (b) After arc-consistency.

Figure 3.21: Arc-consistency reducing symmetry.

Example 32. Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, \{x \geq y, x \neq y, z > y\})$. While arc-consistency will eliminate value 1 from D_z and 3 from D_y (due to constraint $z > y$), the domain of x will remain unchanged after achieving arc-consistency (since all its values are supported), obtaining the arc-consistent CSP $(\{x, y, z\}, \{\{1, 2, 3\}, \{1, 2\}, \{2, 3\}\}, \{x \geq y, x \neq y, z > y\})$. If we choose to represent each constraint of the original CSP by its disallowed assignments, the graph has an automorphism corresponding to the variable symmetry $\langle x, z\rangle \leftrightarrow \langle z, x\rangle$ as shown in Figure 3.21a. However, the graph associated with the arc-consistent CSP no longer has this (or any) variable symmetry as shown in Figure 3.21b. □

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

Let us now consider n-ary arc-consistency.

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

Proof. Let $lit(x_i)$ be the set of literals associated with a variable $x_i \in X$. By assumption, f is a variable or value symmetry and, therefore, $\forall x_i \in X, \exists x_j \in X$ such that $lit(x_j) = f(lit(x_i))$ and $lit(x_i) = f(lit(x_j))$. Also, $\forall c \in C$, each assignment A over $vars(c)$

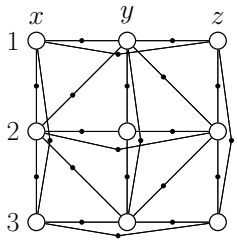


Figure 3.22: CSP with rotational symmetry.

has an image assignment $f(A)$ over the variables $\{f(x) : x \in vars(c)\}$, which we will write $f(vars(c))$. Since each assignment in c has an image over the same set of variables $f(vars(c))$, and since each constraint has a different scope, we can call $f(c)$ the unique constraint over $f(vars(c))$. This constraint has the same number of tuples as c. Also, if c is an allowed constraint then so is $f(c)$, and if c is a disallowed constraint then so is $f(c)$. Note, finally, that since f is a one-to-one mapping of constraints, each constraint $c \in C$ is the image of another constraint $c' \in C$, i.e., $c = f(c')$.

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

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

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

Lemma 6. Arc-consistency does not necessarily preserve all variable-value symmetries.

Proof. We prove this by showing an example of a symmetry that is present before arc-consistency is reached but not afterwards. Consider the CSP $(\{x, y, z\}, \{1, 2, 3\}, C)$, where $C = \{x \neq y, y \neq z, x \neq z, con(x, y), con(z, y)\}$, and con is defined by the following disallowed assignments: $\{\langle 2, 1\rangle, \langle 2, 3\rangle\}$.

The disallowed assignments graph, illustrated in Figure 3.22, admits the rotational symmetry: $\langle x = 1, x = 2, x = 3, y = 1, y = 2, y = 3, z = 1, z = 2, z = 3\rangle \leftrightarrow \langle z = 1, y = 1, x = 1, z = 2, y = 2, x = 2, z = 3, y = 3, x = 3\rangle$. The literal node $x = 2$ can be removed because it is incompatible with every value of the variable y. However, the node $y = 1$ cannot be removed because it is compatible with $x = 3$ and with $z = 3$. Our pruning procedure only removes two literal nodes $x = 2$ and $z = 2$ from the graph, and the disallowed assignments that contain them. As a result, the pruned graph no longer has the rotational symmetry exhibited by the original graph. □

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

Example 33. Consider the CSP (X, D, C) where $X = \{s_1, s_2\}$, $D_{s_1} = D_{s_2} = \{\{\}, \{1\}, \{2\}, \{1, 2\}\}$ (that is, $s_1, s_2 \subseteq \{1, 2\}$) and $C = |s_1 \cap s_2| = 1$. The graph of this CSP using the full assignment graph and allowed assignment nodes is shown in Figure 3.23. As can be seen in the figure, none of the assignments that satisfy the constraint involves $s_1 = \{\}$ or $s_2 = \{\}$. The literal nodes associated with these literals can thus be removed from the graph. □

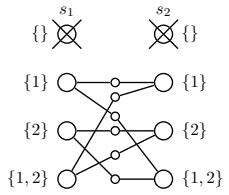


Figure 3.23: Pruning unnecessary values.

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

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

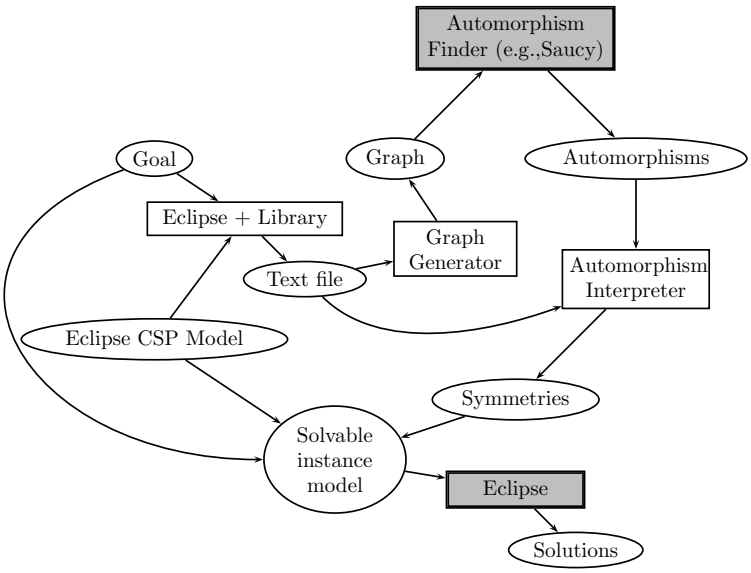


Figure 3.24: System design.

3.6 Experimental evaluation

In this section we evaluate our full assignments graph construction experimentally by applying it to several benchmark problems. We compare the results with our implementation of Puget's variable graph using extensional constraints, and Puget's boolean graph construction. We have extended Puget's representations to handle sets for the purpose of this comparison. The aim of the experiments is to verify that our method can find as many symmetries as possible and that it runs in a reasonable amount of time.

3.6.1 Implementation

We have implemented an automatic symmetry detection system for the subset of ECL^{i}PS^e programs (Apt and Wallace, 2006) that only use finite domain and/or set constraints. The main components of this system are depicted in Figure 3.24, with ovals representing input/output files, white rectangles representing system components, and shaded rectangles indicating external components used by the system.

The first component is an ECL^{i}PS^e library that receives as input the ECL^{i}PS^e program specifying the CSP $P = (X, D, C)$ and outputs a text file containing the set of (syntactic) constraints that would be posted to the solver during the execution of the program. In this file is a line for each variable for each variable x_i in X, indicating its name and domain D_i, and a line for each constraint in C.

This file is, in turn, processed by a graph generator that produces, three possible graph representations of the CSP: the full assignments graph, Puget's variable graph construction using extensional representations of constraints, and Puget's boolean graph construction. This is done as follows. For the full assignments graph, equality constraints are represented by their allowed assignments, disequality constraints by their disallowed assignments, the all-different constraint is split into the equivalent conjunction of binary disequalities, sets

are represented using the extensional representation, and cardinality constraints are represented using their allowed assignments. No other kinds of constraints are needed to represent all our benchmarks. For Puget's extensional representation, equality, disequality, sets and cardinality constraints are represented as before, the all-different constraint is decomposed into disequalities, expressions of the form $op(x)$ where x occurs only once are treated specially, and any other expression is treated using temporary variables as described in Section 3.2.2. Finally, for Puget's boolean representation, each benchmark is converted into a boolean representation as indicated by Section 3.2.1 and all-different constraints are handled using the special representation also described in that section.

For all representations, set variables whose cardinality is constrained to be a constant have their domains pruned so that only those values with the correct cardinality are kept in the graph; in this case, pruning reduces the graph significantly. In other cases no pruning is done as it may reduce the number of variable-value symmetries found as shown in Lemma 6.

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

The resulting graph is input to the graph automorphism package Saucy (Darga et al., 2004) which returns the generating set of the automorphism group. One minor point must be considered: graph automorphism packages consider a graph to be labelled by non-negative integers, whereas our graph nodes have more descriptive labels. Therefore, our system creates a map from graph labels (e.g. $x = 2$) onto integers. This map is also used to convert the numeric labels of the automorphisms found back into descriptive graph labels and allows the detected symmetries to be represented in a more intuitive form for the user.

Example 34. Consider the literal nodes of the graph shown in Figure 3.25 for the 4-queens problem. The label $(x_i = d_i)$ of each literal node is mapped to the positive integer shown within each node (for simplicity, the mapping for assignment nodes is omitted). For this graph, the output of Saucy (omitting the assignment nodes) is:

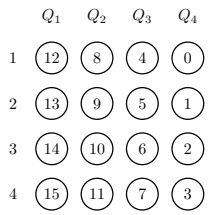


Figure 3.25: Literal nodes of 4-queens.

```
(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 represents one symmetry and each pair of numbers represents a swap of nodes. The first line corresponds to the diagonal variable-value symmetry:
$\langle Q_4 = 2, Q_4 = 3, Q_4 = 4, Q_3 = 3, Q_3 = 4, Q_2 = 4, Q_3 = 1, Q_2 = 1, Q_1 = 1, Q_2 = 2, Q_1 = 2, Q_1 = 3\rangle$
$\leftrightarrow$
$\langle Q_3 = 1, Q_2 = 1, Q_1 = 1, Q_2 = 2, Q_1 = 2, Q_1 = 3, Q_4 = 2, Q_4 = 3, Q_4 = 4, Q_3 = 3, Q_3 = 4, Q_2 = 4\rangle$.

This symmetry is the reflection of the chessboard around the diagonal that runs from the top-right to the bottom-left of the board.

The second line corresponds to the value symmetry $\langle 1, 2\rangle \leftrightarrow \langle 4, 3\rangle$, for each queen. These two generators can be composed to form the group that represents the eight symmetries of a square. □

Although we conducted most of our experiments using Saucy to find graph automorphisms, our implementation is not tied to any particular package. Any graph automorphism package could be used in its place, such as Nauty (McKay, 1981) or AUTOM (Puget, 2005a). For instance, we have successfully tested Nauty with our implementation. We would have liked to use the faster AUTOM (Puget, 2005a), but it is not publicly available.

Section 3.4.5 described two possible ways of representing set variables: using an extensional and a boolean representation. We use the former when evaluating the extensional meaning of constraints and when pruning, and the latter when producing a graph to be searched for automorphisms. This is because the latter yields automorphisms which reflect permutations of the possible elements rather than of the possible sets themselves, a form more suitable to be used as input to symmetry breaking packages such as GAP-SBDS (Gent et al., 2002).

Once the symmetries of a CSP have been found, they can be used to aid a search for the CSP's solutions. While the symmetries detected by our system are only of interest to us as stepping stones towards model-based symmetries (as described in Chapter 4), we wanted to connect our current system to ECL^{i}PS^e and GAP-SBDS to make sure everything was working as expected. The automatic coupling of symmetry detection and symmetry breaking is complicated by the distinction between model and instance. Since the symmetries detected by our system are not applicable to the model, the system creates a new program composed of the original model, the goal that specifies the parameter values used as data for this instance, the symmetries that apply to the instance, and the search predicate that will be used to find a solution. This program can then be executed in ECL^{i}PS^e to solve the CSP instance. Note that, as in most symmetry breaking systems, only unique solutions – those that are not symmetrically equivalent to other solutions – are found.

3.6.2 Benchmarks

Let us now provide a brief summary of the set of benchmarks used in our experimental evaluation. In doing this we will follow the descriptions given in CSPLib (Gent and Walsh, 1999). See Appendix A for more details.

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

We model this problem as a $v \times b$ binary matrix, with constraints that force exactly r ones per row, k ones per column, and a scalar product of λ between any pair of distinct

rows. The symmetries found by all three implemented methods – the full assignments graph, Puget's variable graph construction and Puget's boolean construction – are:

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

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

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

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

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

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

We model this problem using m integer variables and one integer variable per pairwise difference. The difference variables must be all different. A single symmetry is found using the full assignments graph and Puget's boolean graph construction, but no symmetry is found with Puget's variable graph construction. The symmetry corresponds to a 180° reflection of the ruler. This is a variable symmetry on the difference variables, and a variable-value symmetry on the marks variables.

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

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

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

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

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

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

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

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

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

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

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

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

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

- the symmetries of the chessboard (variable symmetry)

- all colours are interchangeable (value symmetry)

Graceful graph: The graceful graph problem is to find a labelling f of the vertices of a graph such that f assigns each vertex a unique label from $\{0, 1, \ldots, e\}$ (where e is the number of edges in the graph), and with each edge (a, b) labelled by $|f(a) - f(b)|$, all the edges labels are different. This benchmark is listed in the results as "graceful-m-n" for the graph $K_m \times P_n$. The symmetries found by all three implemented methods are:

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

3.6.3 Results for symmetry detection

Tables 3.1 and 3.2 show the results of our experimental evaluation of the automatic symmetry detection tool. Each row in the tables corresponds to a different instance of a benchmark problem described in the previous section. A bold font indicates the best result for that row in the table.

The columns in Table 3.1 compare the total number of nodes (Nodes) and the total number of edges in the graph (Edges) when using our implementations of Puget's extensional method (Puget's (Ext)), Puget's boolean method (Puget's (Bool)) and our method (Ours).

The columns in Table 3.2 show the total running time in seconds (Total), followed by a breakdown (expressed as a proportion of the total time) indicating where this time is spent. In particular, the table shows the proportion of time spent in graph generation including the computation of the extensional constraints plus the time spent printing the graph to be input to Saucy (Gr) and the proportion of time taken to read Saucy's output information and print our human-readable form (HR). Any time not accounted for by these two columns is spent running Saucy, and is usually small in comparison. Again, there are three sets of data; one for Puget's extensional method (Puget's (Ext)), one for Puget's boolean method (Puget's (Bool)) and one for our method (Ours). Running times were measured on a desktop with a 3GHz Intel Pentium 4 CPU and 2 GB RAM, running Linux kernel 2.4.22.

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

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

Instance	Graph Details					
	Puget's (Ext)		Puget's (Bool)		Ours	
	Nodes	Edges	Nodes	Edges	Nodes	Edges
bibd-3-3-1-1-0	216	477	216	477	**141**	**297**
bibd-6-10-5-3-2	23737	243576	23737	243576	**3857**	**26730**
golf-2-2-2	1722	4670	1722	4670	**1034**	**2640**
golf-2-2-3	25242	73854	25242	73854	**10650**	**29344**
golf-2-3-2	62841	184515	62841	184515	**24762**	**68109**
golf-3-2-2	4245	11667	4245	11667	**2703**	**7374**
golomb-4	1245	3332	**1006**	**2708**	2484	5456
golomb-5	4815	13505	**3670**	**10380**	9380	21250
golomb-6	14658	42072	**10809**	**31272**	27978	64422
golomb-7	37632	109424	**27181**	**79583**	70665	164297
graceful-3-2	1626	4380	**1085**	**3000**	2235	5070
graceful-5-2	27160	78520	**17897**	**52520**	38155	91390
latin-10	11000	28000	**1300**	**3000**	14500	27000
latin-11	15972	41261	**1694**	**3993**	21296	39930
latin-12	22464	58752	**2160**	**5184**	30240	57024
latin-13	30758	81289	**2704**	**6591**	41743	79092
latin-14	41160	109760	**3332**	**8232**	56252	107016
mostperfect-4	-	-	-	-	**80704**	**314112**
nnqueens-4	460	976	**152**	**304**	464	800
nnqueens-5	1110	2525	**270**	**605**	1175	2100
nnqueens-6	2282	5436	**432**	**1056**	2496	4560
queens-10	1265	3160	**154**	**396**	1570	2940
queens-20	9730	26620	**514**	**1596**	12940	25080
queens-30	32395	91380	**1074**	**3596**	44110	86420
queens-40	76260	218440	**1834**	**6396**	105080	206960
steiner-5	3282	9276	3282	9276	**2115**	**5904**
steiner-6	41975	123090	41975	123090	**22440**	**65850**
steiner-7	347760	1032689	347760	1032689	**154294**	**459557**

Table 3.1: Graph sizes.

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

Table 3.2: Running times.

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

Table 3.3: Running times to find all solutions with and without SBDS.

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

3.6.4 Results for symmetry breaking

As mentioned before, we used the symmetries detected by our implementation to automatically break symmetries during search, using the GAP-SBDS (Gent et al., 2002) library for ECL^{i}PS^e. Results of the experiments for symmetry breaking are shown in Table 3.3. For each benchmark, times are shown for finding all solutions with and without SBDS. The numbers in parentheses show the ratio obtained by dividing the time with SBDS by the time without SBDS. Note that the times shown in Table 3.3 do not include the time needed to find the symmetries. We have omitted these times because the techniques developed in this thesis are ultimately targeted towards finding symmetries in *models* - though in this chapter we only find symmetries in problem instances. With a symmetry detection method that operates on models, the time to find the symmetries can be amortised over all the problem instances in the class defined by the model, and therefore will be small in comparison with problem-solving time.

Note that the aim of this section is not to demonstrate that speedups can be achieved by symmetry breaking methods – this has been the subject of much other work, for example by Freuder (1991), Puget (2005c) and Ramani and Markov (2004) – but, rather, to show that our system is implemented and that the output of our symmetry detection tool can

be easily integrated with a symmetry breaking method, such as GAP-SBDS. Still, we can conclude from the results shown in Table 3.3 that GAP-SBDS performs much better than a simple search when finding all solutions for more than half the benchmarks, with most speedups being of several orders of magnitude. For the rest, the overhead of symmetry breaking is greater than the time saved by reducing the search space. Reducing this overhead is one of the main aims of Chapter 5.

3.7 Conclusion

This chapter has explored the automatic detection of symmetry in CSPs based on graph automorphism. Symmetry detection by graph automorphism can, in theory, be done quite simply: construct a graph with a node for each literal of the CSP and an edge for every set of literals that is disallowed by some constraint or disallowed by being literals of the same variable. This approach is simple but can result in large graphs since a simple constraint, such as an arithmetic equation over several variables, may require very many disallowed edges in its representation. We have examined in detail a method of graph construction described by Puget (2005a), that represents constraints more flexibly. This flexibility can be exploited to keep the graph small, but it could lead to either a failure to capture some of the problem's symmetries, or – worse still – the claimed detection of symmetries that are not truly symmetries of the problem. In order to address this potential drawback, we have identified the conditions under which incorrect results may arise and developed a new graph representation for automatic detection of symmetries.

Too few symmetries may be represented by a graph construction if the graph includes a node for each variable, so that only combinations of value and variable symmetries can be represented and other variable-value symmetries cannot be found. Therefore, we have described a graph representation without such nodes. The drawback of such a representation is that a graph automorphism could map sets of nodes representing solutions to the original problem onto sets of nodes which do not represent a solution, because they require a variable to simultaneously have more than one value. Accordingly, we imposed sufficient conditions on the new graph representation to preclude such automorphisms.

Too few symmetries may also be captured if the graph distinguishes different kinds of constraints. This would prevent, for example, a disequation from being involved in a symmetry with an all-different constraint. To maximise the number of potential symmetries, the graph representation introduced here made no distinction between different constraints. Moreover, it made no distinction between an edge connecting two literals explicitly allowed by a binary constraint, or allowed because their associated variables do not belong to the scope of any constraint. Similarly, it made no distinction between an edge connecting two literals explicitly disallowed due to a constraint, or disallowed because they represent distinct values for the same variables. The main drawback of such a representation is the sheer size of the resulting graph: in principle an allowed edge is required for every compatible set of nodes, and a disallowed edge between every incompatible set. To mitigate this problem, we introduced a full assignments graph representation that uses as few edges as possible without significantly reducing the number of potential symmetries. Since this requires all constraints to be represented extensionally, keeping the size of the

graph as small as possible is very important, particularly for variables with large domains such as set variables.

Too many symmetries are represented by a graph if it has automorphisms that do not correspond to symmetries of the CSP. This can arise, for example, if the existence of an edge does not have a unique meaning. For example, the existence of two edges representing the only two allowed tuples for a single constraint means that *one* should be in a solution, while the existence of two edges representing the only allowed tuples from two different constraints means that *both* should be in a solution. Also, the absence of an edge should have a unique meaning, such as that the two unconnected nodes are unconstrained, or that they are incompatible (e.g. if they represent two different values for a variable). The full assignments graph introduced here is constructed in such a way that the meaning of an edge, or the absence of an edge, is unambiguous and it was proved that every graph automorphism corresponds to a problem symmetry.

This chapter also introduced two additional ways of reducing the size of the graph representation. The first approach is to achieve arc-consistency on the original problem, and build a graph representing this reduced problem. It is shown that achieving arc-consistency by reducing the domains of the variables preserves the variable and value symmetries detected by our approach. However, it also gave an example to show that non-compositional variable-value symmetry may be lost as a result of achieving arc-consistency. The second approach is to represent a constraint by a logically equivalent conjunction of constraints, each with a smaller scope. This was shown to reduce the size of the graphical representation, and potentially improve pruning.

In summary, when compared to Puget's approach, the full assignments graph is simpler but potentially more verbose than the combination of Puget's different graph construction approaches. Indeed, the full assignment graph admits only allowed and disallowed extensional constraints. The benefit is that we have been able to eliminate constraint nodes from the full assignment graph, so as to be able to capture non-composable variable-value symmetries and, at the same time, obtain proofs that graph automorphisms correspond to symmetries of the CSP even when combinations of allowed and disallowed constraints are represented. Puget never attempted such a proof and, without the restrictions introduced in Section 3.4.4, it is shown that certain combinations would lead to graphs whose automorphisms did not correspond to symmetries of the CSP. Moreover, and perhaps surprisingly, it is shown that the extensional graph representation does not necessarily lead to larger graphs than the intensional representation for the boolean model: even for the all-different constraint exampled by Puget, there is an extensional representation with comparable size.

The method of graph construction described in this chapter has been implemented in the constraint logic programming system ECL^{i}PS^e to automatically detect symmetries in a CSP program. The CSP is automatically transformed into a graph, the automorphisms are found using the Saucy package and expressed as constraint symmetries of the CSP, and the CSP is solved using the discovered symmetries to automatically prune the search tree. Experiments were performed on a range of benchmarks to establish the correctness of the implementation.

The approach presented in this chapter is designed to be able to detect as wide a class of constraint symmetries as possible. The key drawback is that it can be used only on

very small problem instances. The efficiency of the symmetry detection method has been a secondary consideration, since we do not plan to use it to detect and apply symmetries for each new problem instance. Rather, the aim is to detect as many representational symmetries as possible for several problem instances, so that they can be tested against a generic problem model, to determine which ones hold for the whole problem class. The key outcome is that the new graph representation presented here finds all of the symmetries that Puget's best graph construction can find, and does not require specially crafted representations of global constraints to do so. These symmetries can then be used to accelerate the solving of any instance of the model. Accordingly, the cost of detecting symmetries can be amortised across all the instances of the problem which are eventually solved using the detected symmetries. Further, the resulting model-based approach will be able to scale up the applicability of our automatic symmetry detection system, since the detected symmetries can be used to accelerate the solving of large practical problems involving hundreds or thousands of variables and constraints.

Chapter 4

Model Symmetry Detection

4.1 Introduction

As discussed in the previous chapter, there has been much work done on the detection of symmetries in constraint satisfaction problems. Almost all of this research has been for individual CSPs, where the exact set of variables, their domains, and the set of constraints are known. However, the utility of such work is limited because the symmetries must be found separately for each CSP, a costly process whose overhead may entirely nullify the savings achieved by exploiting the symmetries found. The time taken in detection of symmetries would be easier to offset if the results applied not only to the problem at hand, but to many other similar problems.

For this reason we explore the detection of symmetries in constraint satisfaction problem *models*. As mentioned in Chapter 1, a CSP model is a parametrised form of CSP, where the overall structure of the problem is given but particular details, such as the size of the problem or a cost threshold, are omitted. These missing data are given as parameters to the model so that the model and a particular parameter set combine to give a CSP. In this way, the model represents a *class* of CSPs, while the model and appropriate data combine to specify an *instance* of that class (i.e. a CSP). We will use the term "CSP instance" to refer to an instance of a CSP model.

For example, recall the Latin square problem of size N (see Section A.8) which is to find an $N \times N$ matrix of values from 1 to N such that each value occurs exactly once in each row and exactly once in each column. The Latin square problem of size 3 can be represented as a CSP that has 9 integer variables $\{x_{11}, x_{12}, x_{13}, x_{21}, \ldots, x_{33}\}$, where x_{ij} represents the cell in row i and column j, each with domain 1..3, and 18 disequality constraints resulting from having 3 constraints for each row and column; for example, in the top row $x_{11} \neq x_{12}$, $x_{11} \neq x_{13}$ and $x_{12} \neq x_{13}$. Alternatively, the problem can be separated into two parts: first, a model that is parametrised on the board size N (with N^2 variables, each with domain $1..N$, and $6(\frac{N^2-N}{2})$ disequality constraints), and second, the data part that simply indicates $N = 3$. Different instances of the class – different CSPs – can be obtained with the same model simply by modifying the value of N in the data.

As far as we known, only two methods for detecting symmetries in CSP models have been proposed. The method by Van Hentenryck et al. (2005) works by having knowledge of the symmetries of individual constraints, and then constructing the symmetries of the

whole model by composing the symmetries of the constraints. This approach can detect only a relatively small set of "simple" symmetries (piecewise value and piecewise variable interchangeability) and is strongly dependent on the particular syntax used to specify the model. The approach by Roy and Pachet (1998) finds structures – a subset of the model's variables and constraints – within the CSP model that are symmetric. It is possible to detect the symmetries among these structures automatically, as one would detect the symmetries of a CSP instance. However, identifying the structures of a problem is not done automatically and requires some problem-specific effort from the user. We propose a radically new approach that:

1. uses accurate symmetry detection methods on a series of small CSP instances to elicit candidate symmetries,

2. parametrises these candidate symmetries to be defined on the model rather than on a particular CSP, and

3. determines whether these candidates are indeed symmetries of the model.

The remainder of this chapter will explain our approach in detail, discuss the results of our experimental evaluation of the framework and explore how the approach is widely applicable to useful properties other than symmetries. The results of our evaluation show that the approach is capable of finding almost all symmetries of a set of benchmark problems in a practical amount of time. Parts of this chapter have previously been published in (Mears et al., 2008b).

4.2 Background

As mentioned before, we are only aware of two automatic symmetry detection methods that deal with CSP models rather than CSP instances. One approach explored by Van Hentenryck et al. (2005) involves defining the symmetries of individual constraints, and then combining these symmetries to obtain the symmetries of the model. This method is based on the fact that many commonly-used constraints are symmetric. For example:

1. simple equality and disequality: $x_1 = x_2$ or $x_1 \neq x_2$. In these constraints both variables are treated identically and therefore there is a variable symmetry $x_1 \leftrightarrow x_2$, and all values are treated identically and therefore all pairs of values are interchangeable.

2. linear constraints, e.g. $x_1 + x_2 + 2x_3 = d$. In such a constraint, variables with the same coefficient are treated identically. Therefore, in this example there is the variable symmetry $x_1 \leftrightarrow x_2$.

3. various global constraints, e.g. $\mathit{all\_different}(x_1, \ldots, x_n)$, where all x_i are treated identically and therefore all pairs of variables are interchangeable.

The key observation that allows symmetries to be derived by composition is the following (proposition 1 in (Van Hentenryck et al., 2005)):

A value-interchangeable CSP is a CSP where all values are interchangeable. (Recall, from Section 3.2, that two values a and b are interchangeable if a can be substituted for b and vice versa in any assignment A without affecting whether A is a solution.) Let $P_1 = (V, D, C_1)$ and $P_2 = (V, D, C_2)$ be two value-interchangeable CSPs. Then, their composition $P_1 \wedge P_2$ is value-interchangeable.

The above proposition can be illustrated by an example.

Example 35. Let $C_1 = \mathit{all\_different}(\{a, b, c\})$ and $C_2 = (d \neq e)$. Both C_1 and C_2 are value-interchangeable, so $C_1 \wedge C_2$ is value-interchangeable. □

Although the above proposition is defined only for the case in which all values are interchangeable, similar results apply when only some values of the problem are interchangeable and also for certain kinds of variable-interchangeability, such as row- and column-interchangeability on matrices of variables.

Unfortunately, this method is limited in its application. A crucial drawback is that symmetries are not inherently compositional: two sub-problems P_1 and P_2 may have no symmetry when considered independently, yet $P_1 \wedge P_2$ may contain symmetries. This kind of symmetry cannot be captured by compositional derivation.

Example 36. Let x and y be finite integer variables with the same domain D. Let $P_1 = (\{x, y\}, D, \{x - y < 5\})$ and $P_2 = (\{x, y\}, D, \{y - x < 5\})$. Taken alone, neither P_1 nor P_2 has symmetries, but in the composition $P = (\{x, y\}, D, \{x - y < 5,\ y - x < 5\})$ the two variables are interchangeable. □

The authors note that many problems can be expressed using global constraints, and such constraints are often symmetric. This mitigates the above drawback in some cases but does not help for problems where symmetries are introduced by the composition of constraints, or where global constraints are not (or cannot be) used. Global constraints may not be available in the modelling language or constraint solver used, the problem may be more easily expressed without them, or the modeller may not have the necessary expertise to use them.

In contrast to this approach, the method of Roy and Pachet (1998) attempts to reduce a CSP model to a CSP instance and then apply a instance symmetry detection method. A model M is transformed into a CSP M' by identifying structures of the model, each of which is represented by a single variable in M'. Then, the symmetries of the CSP M' correspond to symmetries of the model M. However, the difficult problem of identifying the structures of the model is left to the user.

4.3 Running Example: The Latin Square Problem

This section describes an example that will be used to illustrate concepts throughout the chapter. We begin with a concrete CSP instance of the Latin square problem, and show how the symmetries of the instance can be used to find the symmetries of the model.

Example 37. The CSP for the *Latin square* problem of size 3 can be defined as follows. There is one variable x_{ij} for each cell in row i and column j, as shown in Figure 4.1. The domain of each variable is $\{1, 2, 3\}$ and constraints ensure that each value occurs exactly once in each row and exactly once in each column.

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

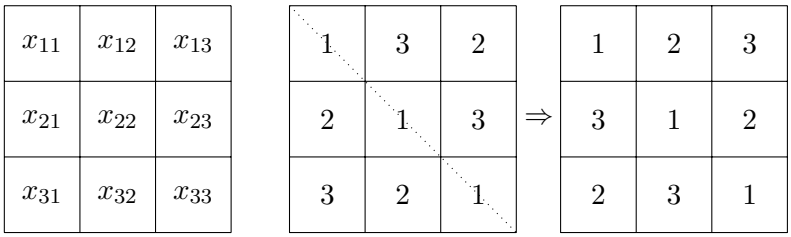


Figure 4.1: Variables in the Latin square and the effect of a diagonal symmetry.

□

Example 38. The Latin square CSP of Example 37 has

- variable symmetries that swap any two columns: $\langle x_{11}, x_{21}, x_{31}\rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32}\rangle$, $\langle x_{11}, x_{21}, x_{31}\rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33}\rangle$, and $\langle x_{12}, x_{22}, x_{32}\rangle \leftrightarrow \langle x_{13}, x_{23}, x_{33}\rangle$
- similar variable symmetries that swap any two rows, and
- variable-value symmetries that transpose any two of the rows, column and value dimensions.

The full assignments graph (left hand side of Figure 4.2) has $9 \times 3 = 27$ nodes (labelled $\begin{matrix}[i,j] \\ k\end{matrix}$) representing the 27 literals $x_{ij} = k$ where $i, j, k \in [1..3]$, and $18 \times 3 + 9 \times 3$ edges representing the 3 assignments disallowed by each of the 18 constraints, and the 3 extra edges needed to disallow each pair of values of the 9 variables. Each variable of the problem can be seen in the Figure as a line of three nodes perpendicular to the front-most face. The front-most face of the cube has, for each variable x, a node representing the literal $x = 1$. Note that much of the graph is omitted for the purpose of legibility. □

As discussed in detail in the previous chapter, the automorphisms of the full assignments graph correspond to symmetries of the CSP.

Example 39. For the Latin square graph of size 3 given in Example 38, the automorphism detection tool Saucy (Darga et al., 2004) returns the following set of generators (illustrated in the left hand side of Figure 4.2):

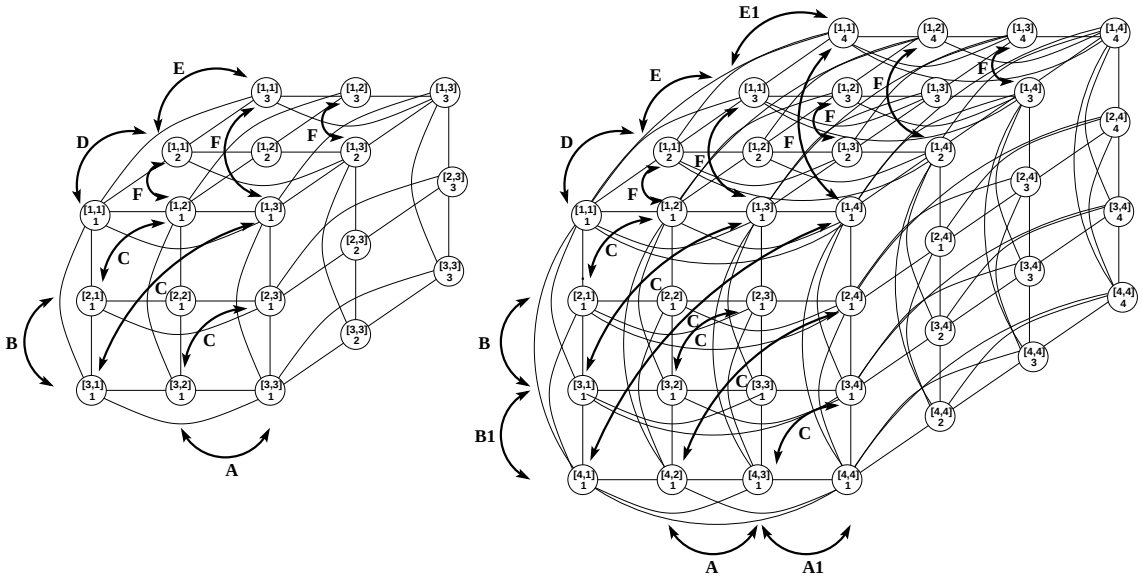


Figure 4.2: Full assignment graphs and generators for LatinSquare[3] and LatinSquare[4]. Note that parts of the graph are obscured.

A $\langle n_{121}, n_{122}, n_{123}, n_{221}, n_{222}, n_{223}, n_{321}, n_{322}, n_{323}\rangle \leftrightarrow$
$\langle n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}, n_{331}, n_{332}, n_{333}\rangle$
B $\langle n_{211}, n_{212}, n_{213}, n_{221}, n_{222}, n_{223}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow$
$\langle n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}, n_{331}, n_{332}, n_{333}\rangle$
C $\langle n_{121}, n_{122}, n_{123}, n_{131}, n_{132}, n_{133}, n_{231}, n_{232}, n_{233}\rangle \leftrightarrow$
$\langle n_{211}, n_{212}, n_{213}, n_{311}, n_{312}, n_{313}, n_{321}, n_{322}, n_{323}\rangle$
D $\langle n_{111}, n_{121}, n_{131}, n_{211}, n_{221}, n_{231}, n_{311}, n_{321}, n_{331}\rangle \leftrightarrow$
$\langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{332}\rangle$
E $\langle n_{112}, n_{122}, n_{132}, n_{212}, n_{222}, n_{232}, n_{312}, n_{322}, n_{333}\rangle \leftrightarrow$
$\langle n_{113}, n_{123}, n_{133}, n_{213}, n_{223}, n_{233}, n_{313}, n_{323}, n_{333}\rangle$
F $\langle n_{112}, n_{113}, n_{123}, n_{212}, n_{213}, n_{223}, n_{312}, n_{313}, n_{323}\rangle \leftrightarrow$
$\langle n_{121}, n_{131}, n_{132}, n_{221}, n_{231}, n_{232}, n_{321}, n_{331}, n_{332}\rangle$

where node n_{ijk} represents literal $x_{ij} = k$. Generator **A** states that columns 2 and 3 can be swapped, **B** that rows 2 and 3 can be swapped, **C** that the square can be reflected across the top-left/bottom-right diagonal, **D** that values 1 and 2 can be swapped, **E** that values 2 and 3 can be swapped, and **F** that the second dimension of the square can be swapped with the value dimension. Their combination results in the symmetries detailed in for Example 38 (e.g., to swap columns 1 and 2 ($\langle x_{11}, x_{21}, x_{31}\rangle \leftrightarrow \langle x_{12}, x_{22}, x_{32}\rangle$) apply first **F**, then **D**, and then **F**).

The generators found for the size 4 version of the problem (right hand side of Figure 4.2) are similar. That is, they result in symmetries that have the same meaning as those described in Example 38, but are not always identical. For example, the symmetry that swaps the first two rows will map n_{141} to n_{241} in the size 4 instance, but not in the size 3 instance since those nodes do not even exist for the smaller size. Also, in the size 4

problem two additional symmetries are found: **A1**, which swaps columns 3 and 4, and **E1**, which swaps values 3 and 4. □

4.4 From CSPs to parametrised CSPs

There is no standard notation to distinguish a CSP instance and a CSP model (or *parametrised CSP*). Herein, we denote a parametrised CSP as $CSP[Data]$, where $Data$ represents the parameters, and a particular CSP instance in that class as $CSP[d]$, where d is the value given to $Data$ to yield that CSP. In a sense, the parametrised CSP $CSP[Data]$ is a function that maps a parameter set onto a CSP instance. While we use mathematical notation to specify parametrised CSPs, any high-level modelling language can be used (such as Zinc (Garcia de la Banda et al., 2006), OPL (Van Hentenryck, 1999), Essence (Frisch et al., 2007), etc.) as long as it separates the model from the data, has multi-dimensional arrays of finite domain variables, and supports iteration over those arrays.

Example 40. The parametrised LatinSquare[N] for the CSP of Example 37 is as follows:

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

This parametrised CSP defines N^2 integer decision variables ($square_{ij}$) with values in $1..N$, and conjoins the disequality constraints for every row i and column j. □

Our aim is to determine the symmetries of every CSP in the class represented by $CSP[Data]$ – in other words, the symmetries of $CSP[d]$, for every d that could possibly be given as $Data$. To formalise this task, we define the *parametrised* graph $G[Data]$ of $CSP[Data]$ in such as way that, when instantiated by giving a value d to $Data$, $G[d]$ yields the full assignments graph of $CSP[d]$. The parametrised graph can be viewed, similarly to the parametrised CSP, as a function that maps a parameter set onto a graph. Formally, $G[Data]$ is obtained from $CSP[Data] = (X[Data], D[Data], C[Data], dom[Data])$ as follows:

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

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

Example 41. The parametrised graph $G[N]$ associated with LatinSquare[N] is as follows. $V[N]$ is defined as $\{n_{ijv}|i,j,v \in [1..N]\}$ where n_{ijv} denotes literal $square_{ij} = v$. $E_v[N]$ is defined as $\{\{n_{ijv_1}, n_{ijv_2}\}|i,j,v_1,v_2 \in [1..N], v_1 \neq v_2\}$, while $E_c[N]$ is obtained by transforming the two constraints in LatinSquare[N] into the set of assignments they disallow:

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

Note that the nodes in $G[N]$ maintain some of the structure of LatinSquare[N] thanks to the reuse of the i and j identifiers appearing in LatinSquare[N]. This is important for the automation of the construction of the edges in $G[N]$ and, as will be discussed later, for the parametrisation of the symmetries of a CSP. □

We can now define a parametrised symmetry.

Definition 5. Given a parametrised $CSP[Data]$ and its parametrised graph $G[Data]$, a parametrised permutation $f[Data]$ is a bijection of the nodes of $G[Data]$. That is, for all values d given to $Data$, $f[d]$ permutes the nodes of $G[d]$. A parametrised symmetry of $CSP[Data]$ is a parametrised permutation $f[Data]$ of the nodes in $G[Data]$ s.t. for all values d given to $Data$, $f[d]$ is a symmetry (an automorphism) of $G[d]$.

Example 42. Two parametrised symmetries of the parametrised LatinSquare[N] CSP of Example 40 are the one that reflects the square from top to bottom and one that reflects around the diagonal. These can be written as:

$$f[N](square_{ij} = k) = (square_{ji} = k)$$
$$f[N](square_{ij} = k) = (square_{(N-i+1)j} = k)$$

Note that these hold for all values of N. □

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

Taken to its extreme, this definition of parametrised CSP allows all problems to be represented by a single model. This model would leave the variables, domains and constraints completely unspecified, and the data is required to fill in all of the details. However, by our definition of parametrised symmetry, this model has only one symmetry – the identity – and is not an interesting example for symmetry detection. As discussed in the next section, our implementation restricts the parameter of a model to be a sequence of integers. That the model symmetries must be present in every instance is a limitation of the framework. For example, if the data is a graph, a symmetry in the graph given as data may result in a symmetry in the CSP. However, if the structure of the data is unknown, our framework cannot find that symmetry. It is possible that more expressive kinds of parameters may still allow useful symmetries to be found, but this is outside the scope of this thesis.

4.5 A framework for detecting parametrised symmetries

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

1. Detect the instance symmetries of $CSP[d]$ for a number of values d given to $Data$,
2. Lift the symmetries found in step one to obtain parametrised permutations of the literals in $CSP[Data]$,
3. Filter the parametrised permutations from step two to keep only those that are likely to be parametrised symmetries,
4. Prove that the parametrised permutations selected in step three are indeed parametrised symmetries.

This section deals with the first three steps in detail. The final step will be discussed in Section 4.8.

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

The first step of our generic framework can be realised in different ways by the choice of parameter values and of symmetry detection method. These choices are somewhat mutually dependent: using a powerful symmetry detection method will usually force the parameter values to be small (e.g. if a graph automorphism method is used, the instance must be small enough for the graph to be computed in a reasonable time). Our implementation uses the detection method described in the previous chapter which returns the group of symmetries in a $CSP[d]$ as a set of group generators. Note that since the instance symmetry detection method is incomplete (i.e., might miss some symmetries), our implementation of the generic framework is also incomplete. As mentioned before, for simplicity our implementation assumes that the parameter $Data$ is a tuple of k integers, $(p_1, p_2, \ldots, p_k)$ and generates different parameter values d by increasing each component of the tuple individually, starting from some user-defined base tuple (typically the smallest meaningful instance of the class).

Example 43. For LatinSquare[N], $Data$ has a single component: the board size N. If the user provides (3) as the base tuple, we increment the component twice obtaining three values for d: (3), (4), and (5). For the social golfers problem (see Section 4.6), $Data$ has three components: the number of weeks, groups per week and players per group. If $(2, 2, 2)$ is the base tuple, we increment twice each component to get nine values for d (seven of which are distinct): $(2, 2, 2), (3, 2, 2), (4, 2, 2), (2, 2, 2), (2, 3, 2), (2, 4, 2), (2, 2, 2)$, $(2, 2, 3)$, and $(2, 2, 4)$. □

4.5.2 Step two: Lifting symmetries to parametrised permutations

This step requires taking every symmetry g detected in step one for any of the $CSP[d]$ considered, and determining one or more parametrised permutation(s) $f[Data]$ for which

$f[d] = g$. Since computing $f[Data]$ from g alone is quite a task, our implementation uses a much simpler, although incomplete, method: it first defines a set of "common" parametrised symmetries $Per = \{f_1[Data], \cdots, f_m[Data]\}$, and then checks every generator g against them.

The success of our implementation relies on the parametrised CSPs having literals that can be arranged into an n-dimensional matrix, and having "common" parametrised symmetries that permute particular matrix elements, such as rows or columns. These are the kind of symmetries that we will add to Per. This limits the kind of symmetries we can consider, but matrix models of problems are very common (Flener et al., 2002b) and these symmetries seem to be those that are most useful in practice (as seen in the next chapter).

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

Example 44. The parametrised LatinSquare[N] problem has a matrix like structure, since its literals can be arranged into a 3-dimensional matrix where each literal $square_{ij} = k$ is indexed as $L[N]_{i,j,k}$. This is clearly visible in Figure 4.2, where the only difference between $G[3]$ and $G[4]$ are the exact values of each dimension: $Dims[3] = (3, 3, 3)$ while $Dims[4] = (4, 4, 4)$. □

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

- **Value swap:** interchanges values v and v' of the k^{th} dimension and is defined as:
  $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow$
  $L[Data]_{i_1,\ldots,i_{k-1},v',i_{k+1},\ldots,i_n}$ $\forall i_j \in [1..d_j], j \in [1..n]$.
  For example, the symmetry represented by generator **D** in Figure 4.2 is a value swap with $k = 3, v = 1, v' = 2$.

- **Dimension invert:** interchanges every value v of the k^{th} dimension with value $n - v + 1$ and is defined as:
  $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow$
  $L[Data]_{i_1,\ldots,i_{k-1},n-v+1,i_{k+1},\ldots,i_n}$, $\forall v \in [1..d_k]$, $i_j \in [1..d_j], j \in [1..n]$.
  For example, the symmetry represented by generator **A** in Figure 4.3 is a dimension invert with $k = 2$.

- **Dimension swap:** swaps k^{th} and k'^{th} dimensions and is defined as:
  $L[Data]_{i_1,\ldots,i_{k-1},i_k,i_{k+1},\ldots,i_{k'-1},i_{k'},i_{k'+1},\ldots,i_n} \leftrightarrow$
  $L[Data]_{i_1,\ldots,i_{k-1},i_{k'},i_{k+1},\ldots,i_{k'-1},i_k,i_{k'+1},\ldots,i_n}$, $\forall i_j \in [1..d_j], j \in [1..n]$.

For example, the symmetry represented by generator **B** in Figure 4.3 is a dimension swap with $k = 1, k' = 2$.

Example 45. The generators found for LatinSquare[3] in Example 39 can be automatically matched to the following parametrised permutations for $L[N]$:

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

Consider the graph $G[4]$ associated with LatinSquare[4], shown in the right hand side of Figure 4.2. Saucy finds 9 generators for this graph. Six of them are simple extensions of those found for $G[3]$. For example, the extension of **A** is:

A $\langle n_{121}, n_{122}, n_{123}, n_{124}, n_{221}, n_{222}, \ldots, n_{321}, \ldots, n_{421}, \ldots\rangle \leftrightarrow$
$\langle n_{131}, n_{132}, n_{133}, n_{134}, n_{231}, n_{232}, \ldots, n_{331}, \ldots, n_{431}, \ldots\rangle$

and similarly for **B, C, D, E** and **F**. The other three generators found are:

A1 $\langle n_{131}, n_{132}, n_{133}, n_{134}, n_{231}, n_{232}, \ldots, n_{331}, \ldots, n_{431}, \ldots\rangle \leftrightarrow$
$\langle n_{141}, n_{142}, n_{143}, n_{144}, n_{241}, n_{242}, \ldots, n_{341}, \ldots, n_{441}, \ldots\rangle$
B1 $\langle n_{311}, n_{312}, n_{313}, n_{314}, n_{321}, n_{322}, \ldots, n_{331}, \ldots, n_{341}, \ldots\rangle \leftrightarrow$
$\langle n_{411}, n_{412}, n_{413}, n_{414}, n_{421}, n_{422}, \ldots, n_{431}, \ldots, n_{441}, \ldots\rangle$
E1 $\langle n_{113}, n_{123}, n_{133}, n_{143}, n_{213}, n_{223}, \ldots, n_{313}, \ldots, n_{413}, \ldots\rangle \leftrightarrow$
$\langle n_{114}, n_{124}, n_{134}, n_{144}, n_{214}, n_{224}, \ldots, n_{314}, \ldots, n_{414}, \ldots\rangle$

The generators **A, B, C, D, E** and **F** in $G[4]$ match the parametrised permutations used for $G[3]$, while **A1, B1** and **E1** match:

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

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

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

□

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

- **All values swap:** interchanges all values of the k^{th} dimension and is defined as:
  $L[Data]_{i_1,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow$
  $L[Data]_{i_1,\ldots,i_{k-1},v',i_{k+1},\ldots,i_n}$, $\forall v, v' \in [1..d_k], v \neq v'$, $i_j \in [1..d_j], j \in [1..n]$.
  For example, the symmetries represented by generators **D** and **E** and their combinations in the left hand side of Figure 4.2 are an all values swap with $k = 3$.

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

Example 46. The generators **D** and **E** for LatinSquare[3] form an instance of the "all value swap" pattern $L[N]_{ijv} \leftrightarrow L[N]_{ijv'}, \forall v, v' \in [1..N], v \neq v'$, $i, j \in [1..N]$. The generators **D**, **E** and **E1** for LatinSquare[4] form the same pattern. □

4.5.3 Step three: Filtering parametrised permutations

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

Example 47. In the case of the Latin square problem, using instance sizes 3, 4 and 5, the following parametrised symmetries are found by our implementation:

- dimensions 1 and 2 swap (diagonal reflection of the square)
- dimensions 2 and 3 swap (column dimension swaps with value dimension)
- all values of dimension 3 are interchangeable (all values interchangeable)

Each of these symmetries is found in every tested instance, so a simple intersection of the symmetries would result in all symmetries being found. □

Unfortunately, the success of such an intersection relies on Saucy returning the same (or equivalent) set of generators for each $CSP[d]$. This is because, as mentioned before, our implementation only attempts to parametrise the generators returned by Saucy (as opposed to every symmetry in the group), and a group can be obtained from many different sets of generators. We solve this problem as follows. If a particular parametrised permutation is found in more than one instance but not in all, we check the group of symmetries of the other instances to see if the permutation is, in fact, present. This is done via the GAP system for computational group theory (Group, 2006). If the parametrised permutation is indeed found in all instances, it is marked as a candidate.

Example 48. Consider the social golfers problem with values of d being $(2,2,2)$, $(3,2,2)$, $(4,2,2)$, $(2,3,2)$, $(2,4,2)$, $(2,2,3)$, $(2,2,4)$. Our implementation finds an instance of the "all value swap" pattern for the third dimension (golfers are interchangeable) for every value of d. However, the "all value swap" pattern for the first dimension (the weeks are interchangeable) is found for only 5 out of the 7 values of d, due to the particular generators given by Saucy. Searching explicitly for this pattern in the groups found for the other values of d shows that it is indeed present in all of them and can thus be considered as a likely candidate. □

4.6 Detailed Examples

We have discussed how the first three steps of our implementation work with the Latin square example. Here we illustrate them further with some detailed examples, and delay the discussion of the fourth step until Section 4.8. We provide three more examples: N-queens, for which our method again detects all parametrised symmetries as likely candidates; Social golfers, for which it also detects all symmetries (after adding a new pattern); and Golomb ruler, for which it fails to detect any likely candidate.

4.6.1 N-queens

The N-queens problem is to position N queens on an $N \times N$ chess board without attacking each other (see Section A.1). The following parametrised CSP NQueens[N] uses N integer variables q_i where $q_i = j$ if a queen appears in row j of column i.

$$\begin{aligned}
X[N] &= \{q_i | i \in [1..N]\} \\
D[N] &= [1..N] \\
C[N] &= \{q_i \neq q_j | i \in [1..N], j \in [i+1..N]\} \cup \\
&\quad \{q_i + i \neq q_j + j | i \in [1..N], j \in [j+1..N]\} \cup \\
&\quad \{q_i - i \neq q_j - j | i \in [1..N], j \in [j+1..N]\}
\end{aligned}$$

The $q_i \neq q_j$ constraint ensures that no two queens share a row, and the other two constraints ensure that no two queens share a diagonal in either direction (rising or falling). The parametrised graph $G[N] = (V[N], E_c[N] \cup E_v[N])$ is:

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

where node q_{iv} represents literal $q_i = v$. Given the initial base tuple (4), our implementation generates $G[4]$, $G[5]$ and $G[6]$. Figure 4.3 shows the full assignment graph for instances $G[4]$ and $G[5]$, together with the generators found by Saucy. Each node in the graph is marked by "qi" and a value j, representing the literal $q_i = j$. Each edge, shown with a thin black line, indicates that the two literals at the end-points are disallowed. The symmetries are marked with thick arrows.

For $G[4]$ the following symmetry generators are found by Saucy:

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

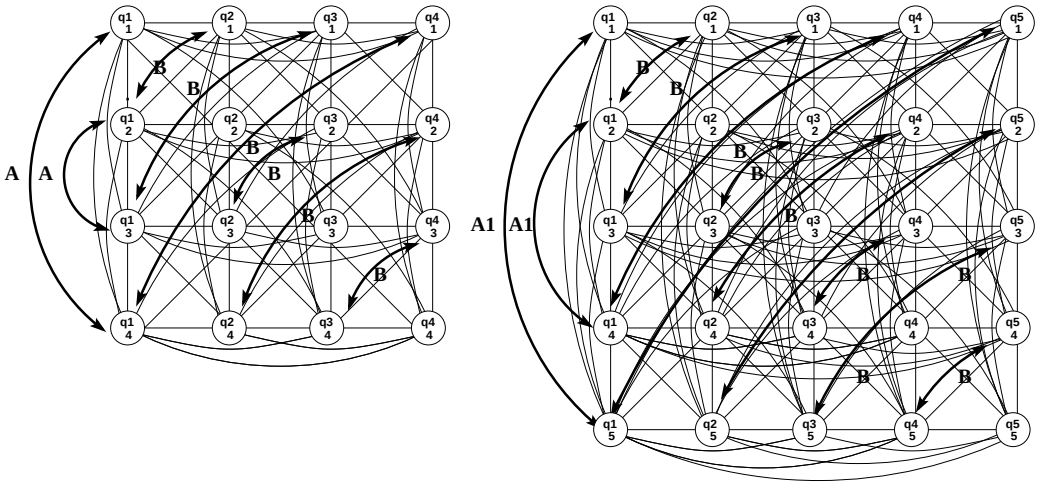


Figure 4.3: Full assignment graphs of instances NQueens[4] and NQueens[5]

Symmetry **A** corresponds to an inversion of the values, mapping each value i to $5 - i$. This is the reflection of the chessboard around the central horizontal axis. The symmetry **B** swaps the variable index and value dimensions, corresponding to a diagonal reflection of the chessboard. These symmetries can be parametrised to match:

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

For $G[5]$ Saucy finds the following symmetry generators:

A1 $\langle q_{11}, q_{12}, q_{21}, q_{22}, q_{31}, q_{32}, q_{41}, q_{42}, q_{51}, q_{52}\rangle \leftrightarrow$
$\langle q_{15}, q_{14}, q_{25}, q_{24}, q_{35}, q_{34}, q_{45}, q_{44}, q_{55}, q_{54}\rangle$
B $\langle q_{12}, q_{13}, q_{14}, q_{15}, q_{23}, q_{24}, q_{25}, q_{34}, q_{35}, q_{54}\rangle \leftrightarrow$
$\langle q_{21}, q_{31}, q_{41}, q_{41}, q_{32}, q_{42}, q_{52}, q_{43}, q_{53}, q_{45}\rangle$

where **B** is an extension of the generator with the same name found for $G[4]$ (and matches the same dimension swap pattern), and **A1** is a new generator that matches the same dimension invert pattern as **A**. The generators found for $G[6]$ are, again, an extension of **B** that matches the dimension swap pattern, and a new generator **A2** that matches the same pattern as **A** and **A1**. The intersection of the patterns results in both parametrised permutations being marked as likely candidates. These parametrised permutations form all of the actual symmetries of the problem.

4.6.2 Social Golfers

The social golfers problem is to build a schedule of W weeks, with G equally-sized groups per week, and P golfers per group, such that each pair of golfers may play in the same group at most once (see Section A.2). A parametrised CSP Golf$[W, G, P]$ is:

$X[W, G, P] = \{players_{wg} | w \in [1..W], g \in [1..G]\}$
$D[W, G, P] = \wp(\{1..P * G\})$

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

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

$$
\begin{aligned}
V[W,G,P] = {} & \{n_{wgp} | w \in [1..W], g \in [1..G], p \in \wp([1..P * G])\} \\
E_c[W,G,P] = {} & \{\langle n_{wgp} \rangle | w \in [1..W], g \in [1..G], |p| \neq P\} \cup \\
& \{\langle n_{wg_1p_1}, n_{wg_2p_2} \rangle | w \in [1..W], g_1, g_2 \in G, g_1 < g_2, \\
& \qquad p_1, p_2 \in \wp([1..P * G]), |p_1 \cap p_2| \neq 0)\} \cup \\
& \{\langle n_{w_1g_1p_1}, n_{w_2g_2p_2} \rangle | w_1, w_2 \in [1..W], w_1 < w_2, g_1, g_2 \in G, \\
& \qquad g_1 < g_2, p_1, p_2 \in \wp([1..P * G]), |p_1 \cap p_2| > 1\} \\
E_v[W,G,P] = {} & \{\langle n_{wgp,p}, n_{wgp_2} \rangle | w \in [1..W], g \in [1..G], p_1, p_2 \in \wp([1..P * G]), p_1 \neq p_2\}
\end{aligned}
$$

where node n_{wgp} represents literal $players_{wg} = p$. The three components of E_c are:

- a unary edge for each literal for which the value's cardinality is not P.
- an edge between any two literals for the same week whose intersection is not null – this ensures that the groups in each week form a partition of the set of players.
- an edge between any two literals for distinct weeks whose intersection is more than one – this is disallowed by the specification of the problem.

For Golf$[2,2,2]$ the following symmetry generators are found by Saucy. For brevity we use a slightly different notation here: the term v_{wgx} represents $players_{wg} \ni x$; i.e. player x is in group g of week w. A value of 0 for x represents an empty set; i.e. v_{110} represents the literal $players_{11} = \emptyset$. This notation also allows the symmetry among the players to be seen more easily.

A $\langle v_{113}, v_{123}, v_{213}, v_{223} \rangle \leftrightarrow \langle v_{114}, v_{124}, v_{214}, v_{224} \rangle$
B $\langle v_{112}, v_{122}, v_{212}, v_{222} \rangle \leftrightarrow \langle v_{113}, v_{123}, v_{213}, v_{223} \rangle$
C $\langle v_{111}, v_{121}, v_{211}, v_{221} \rangle \leftrightarrow \langle v_{112}, v_{122}, v_{212}, v_{222} \rangle$
D $\langle v_{110}, v_{111}, v_{112}, v_{113}, v_{114} \rangle \leftrightarrow \langle v_{120}, v_{121}, v_{122}, v_{123}, v_{124} \rangle$
E $\langle v_{210}, v_{211}, v_{212}, v_{213}, v_{214} \rangle \leftrightarrow \langle v_{220}, v_{221}, v_{222}, v_{223}, v_{224} \rangle$
F $\langle v_{110}, v_{111}, v_{112}, v_{113}, v_{114}, v_{120}, v_{121}, v_{122}, v_{123}, v_{124} \rangle \leftrightarrow$
$\langle v_{210}, v_{211}, v_{212}, v_{213}, v_{214}, v_{220}, v_{221}, v_{222}, v_{223}, v_{224} \rangle$

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

- **Conditional value swap:** interchanges values v and v' of the k^{th} dimension under the condition that the value of another dimension l is some constant x.
  $L[Data]_{i_1,\ldots,x,\ldots,i_{k-1},v,i_{k+1},\ldots,i_n} \leftrightarrow$
  $L[Data]_{i_1,\ldots,x,\ldots,i_{k-1},v',i_{k+1},\ldots,i_n}\ \forall i_j \in [1..d_j], j \in [1..n]$.

The generator **D** is a conditional value swap with $v = 1$, $v' = 2$, $k = 2$, $l = 1$ and $x = 1$. The generator **E** is a conditional value swap with the same parameters, except that $x = 2$.

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

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

Therefore, our implementation marks as likely candidate the following parametrised permutations:

- for each value of dimension 1, the values of dimension 2 are interchangeable – the groups are interchangeable independently for each week.
- values of dimension 1 are interchangeable – the weeks are interchangeable.
- values of dimension 3 are interchangeable – the players are interchangeable.

These parametrised symmetries cover all of the constraint symmetries found at the instance level of the problem.

4.6.3 Golomb ruler

is defined as a set of N integers (marks on the ruler) $a_1, \ldots, a_N$ such that the $\frac{N(N-1)}{2}$ differences $a_j - a_i, 1 \leq i < j \leq N$ are distinct. The problem involves finding a valid set of N marks (see Section A.3). The following parametrised CSP Golomb[N] uses N integer variables (the marks) with domains in $[0..N^2]$, plus $\frac{N(N-1)}{2}$ integer variables (the differences) with domains $[1..N^2]$.

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

The parametrised graph associated with Golomb[N] is:

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

where node m_{iv} represents literal $mark_i = v$ and node d_{ijv} literal $diffs_{ij} = v$. The generator found by Saucy for $G[3]$ is:

$$
\begin{aligned}
\mathbf{A}\ &\langle d_{121}, d_{122}, d_{123}, d_{124}, d_{125}, d_{126}, d_{127}, d_{128}, d_{129}\rangle \leftrightarrow \\
&\langle d_{231}, d_{232}, d_{233}, d_{234}, d_{235}, d_{236}, d_{237}, d_{238}, d_{239}\rangle \text{ plus} \\
&\langle m_{10}, m_{11}, m_{12}, m_{13}, m_{14}, m_{15}, m_{16}, m_{17}, \ldots, m_{24}\rangle \leftrightarrow \\
&\langle m_{39}, m_{38}, m_{37}, m_{36}, m_{35}, m_{34}, m_{33}, m_{32}, \ldots, m_{25}\rangle
\end{aligned}
$$

which swaps the lengths of the spaces between the marks, i.e., turns the ruler back-to-front. This symmetry involves variables from two separate matrices, d_{ij} and m_i, and our simple implementation cannot yet handle this. Even if we only consider the search variables m_i, our implementation would need to obtain for $G[3]$, $G[4]$ and $G[5]$ the pattern $\{m_{iv} \leftrightarrow m_{jv'} | i, j \in [1..N], i = N - j + 1, v, v' \in [0..N^2], v = N^2 - v' + 1\}$. Since our implementation currently does not take this pattern into account, it cannot recognise the permutation as likely candidate. It would not be difficult to add a pattern to match this kind of symmetry. However, if this pattern occurs only rarely then it may not be worthwhile, as each pattern increases the amount of time needed to parametrise instance symmetries.

4.7 Results

Let us evaluate our simple implementation (which includes the patterns described in Section 4.5.2 plus the additional pattern described for Social Golfers) over a set of problems that include those discussed earlier in this chapter, plus the following (see Appendix A for details):

Balanced Incomplete Block Design: with parameters (v, b, k, r, λ), where the task is to arrange v objects into b blocks such that each block has exactly k objects, each object is in exactly r blocks, and every pair of objects occurs together in λ blocks. The objects are interchangeable and the blocks are interchangeable.

Graceful Graph: with parameters (m, n), where the edges (a, b) of the graph $K_m \times P_n$ are labelled by $|a-b|$, and there is no two edges with the same label. The corresponding vertices in each clique are simultaneously interchangeable, the order of the cliques is reversible, and the values are reversible.

N × N queens: where an $N \times N$ chessboard is coloured with N colours, so that a pair of queens in any two squares of the same colour do not attack each other. The symmetries are those of the chessboard, plus the colours are interchangeable.

NQueens (bool): which uses a Boolean matrix model for the N-queens problem of Section 4.6. The symmetries are those of the chessboard.

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

Table 4.1: Symmetry detection results

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

Table 4.1 shows the results, where the columns indicate the problem name, the base tuple, the amount by which each component is increased, the known symmetries and whether they are marked as likely candidates by our implementation, the total running time in seconds, and the percentage of that time spent in detection (as opposed to parametrisation). The experiments were run on an dual Intel Core 2 1.86GHz computer with 1GB of memory. No effort has been made to optimise detection time; the times are included simply to show the practicality of the approach.

The results show that the approach is capable of finding almost all symmetries of this set of problems. The constraints used in each problem vary, from many simple constraints to global constraints. In particular, all symmetries are found in the two quite different representations of the N-queens problem. The times taken are small, due to the small instances used. However, the small instances do not seem to be an impediment to finding all of the symmetries.

4.8 Proving parametrised symmetries

The final step of the framework is to check whether the parametrised permutations marked as likely candidates in step three are truly symmetries of the problem. This step is important if the symmetries are to be later used in symmetry breaking to make the search more efficient, for if the permutations are not symmetries then the search may fail to find some solutions.

In many cases this step can be performed trivially by a human. Although it is often difficult to discover the symmetries of a given problem, it is generally easy for a person who is familiar with the problem to confirm that a given candidate symmetry is correct. Therefore, an automatic method that implements only the first three steps and leaves the

confirmation of symmetries to the user is still useful. However, the ability to automatically prove that symmetries hold over all instances of a model would make the framework even more useful.

The proving of parametrised symmetries can be achieved, for example, by first representing both the parametrised CSP and the candidate parametrised permutation in the logic formalism described by Mancini and Cadoli (2005), and then making use of theorem proving techniques. Of course, such a technique is in general undecidable. This approach suffers from another problem in that a user must find an equivalent logical expression for their constraint program model, a task that may be difficult. In this sense, such an approach is not truly automatic.

We have explored two alternative methods of proving that symmetries hold on a model. Although neither is sufficiently robust to be used generally, they show the direction in which a more general approach to symmetry proving might be found.

4.8.1 Proving via Parametrised Graph

The first approach is to use the parametrised graph of the constraint model to prove that a parametrised permutation is a model symmetry. Since the symmetries of the parametrised graph reflect symmetries of the problem, showing that a parametrised permutation is an automorphism of the parametrised graph is sufficient to prove that it is a symmetry on the model.

Consider the parametrised graph $G[Data]$ of a parametrised $CSP[Data]$. Let $neighbours(n)$ be a function which, given a node $n \in V[Data]$ returns all nodes in $V[Data]$ that are adjacent to n; that is, all nodes u for which there is an edge $(n, u) \in E[Data]$. To prove that a parametrised permutation σ is a model symmetry, we must prove that for all nodes n:

$$\sigma(neighbours(n)) = neighbours(\sigma(n)) \tag{4.1}$$

To prove that (4.1) holds, we must show that every element of the left hand side is also a member of the right hand side. We illustrate how this might be done with an example.

Figure 4.4 shows a simple representation of the N-queens problem and the *neighbours* function for its parametrised graph. Note that the representation of the problem is similar to how the problem might be expressed in a constraint programming language. In this form, the only constraints are binary $\neq$ constraints, although the operands can involve the addition of constants.

The constant N is the parameter to the model, and q is an array of integer variables whose domains are 1 to N. When calculating the *neighbours* function, we represent the literal $q_i = j$ – which is represented as a single node in the graph – by the pair $[i, j]$. As can be seen in Figure 4.4, the neighbours of a node $[i, j]$ are all other literals that involve the same variable (by definition, a variable has only one value) and any literals that are disallowed from occurring with $[i, j]$ by some constraint. These latter neighbours are computed by substituting the variable index i for the indices in the problem, namely x and y. For example, in constraint (1), substituting i for x gives $q_i \neq q_y$, where y is restricted to be $i + 1 \leq y \leq N$. This is the set of nodes denoted by (1a). For example, with $N = 4$, the constraint after substitution is $q_i \neq q_y$, where $i + 1 \leq y \leq 4$ and for all

```
for 1 <= x <= N
  for x+1 <= y <= N
    q[x] /= q[y]          (1)
    q[x]-x /= q[y]-y      (2)
    q[x]+x /= q[y]+y      (3)
```

$$
\begin{aligned}
\mathit{neighbours}([i,j]) = \{\ & [i,k], 1 \le k \le N, k \neq j && \text{(defn. of variable)} \\
& [k,j], i+1 \le k \le N && \text{(1a)} \\
& [k,j], 1 \le k \le i-1 && \text{(1b)} \\
& [k,j-k+i], i+1 \le k \le N && \text{(2a)} \\
& [k,j-k+i], 1 \le k \le i-1 && \text{(2b)} \\
& [k,j+k-i], i+1 \le k \le N && \text{(3a)} \\
& [k,j+k-i], 1 \le k \le i-1 && \text{(3b)} \\
\}
\end{aligned}
$$

Figure 4.4: The N-queens problem.

$1 \le i \le 4$. Similarly, substituting i for y in constraint (1) gives $q_x \neq q_i$, where x must obey $1 \le x \le y-1$. This leads to the nodes denoted (1b). Similar reasoning leads to (2a), (2b), (3a) and (3b).

Expressed in this form, one symmetry of the problem is

$$\sigma([i,j]) = [N-i+1, j] \tag{4.2}$$

This is the reflection of the variables, equivalent to the reflection of the chessboard. We now show how to prove this symmetry. The LHS of (4.1) is $\sigma(\mathit{neighbours}(n))$, which is the symmetry σ applied to the neighbours of $[i,j]$.

$$
\begin{aligned}
\mathit{LHS} = \{\ & [N-i+1,k], k \neq j && \text{(a)} \\
& [N-k+1,j], i+1 \le k \le N && \text{(b)} \\
& [N-k+1,j], 1 \le k \le i-1 && \text{(c)} \\
& [N-k+1,j-k+i], i+1 \le k \le N && \text{(d)} \\
& [N-k+1,j-k+i], 1 \le k \le i-1 && \text{(e)} \\
& [N-k+1,j+k-i], i+1 \le k \le N && \text{(f)} \\
& [N-k+1,j+k-i], 1 \le k \le i-1 && \text{(g)} \\
\}
\end{aligned}
$$

The right hand side is the neighbours of $\sigma([i,j]) = [N-i+1, j]$.

$$
\begin{aligned}
\mathit{RHS} = \{\ & [N-i+1,k], k \neq j && \text{(A)} \\
& [k,j], (N-i+1)+1 \le k \le N && \text{(B)}
\end{aligned}
$$

$$
\begin{aligned}
&[k, j], 1 \leq k \leq (N - i + 1) - 1 && \text{(C)}\\
&[k, j - k + (N - i + 1)], (N - i + 1) + 1 \leq k \leq N && \text{(D)}\\
&[k, j - k + (N - i + 1)], 1 \leq k \leq (N - i + 1) - 1 && \text{(E)}\\
&[k, j + k - (N - i + 1)], (N - i + 1) + 1 \leq k \leq N && \text{(F)}\\
&[k, j + k - (N - i + 1)], 1 \leq k \leq (N - i + 1) - 1 && \text{(G)}\\
\}
\end{aligned}
$$

Now we must show that every element in the LHS is a member of the RHS. (a) is clearly a member of the RHS; it is identical to (A). We can show that (b) is equal to (C), by first manipulating the bounds of (b).

$$
\begin{aligned}
i + 1 \leq \quad & k \quad && \leq N\\
-i - 1 \geq \quad & -k \quad && \geq -N\\
n - i - 1 \geq \quad & N - k \quad && \geq 0\\
n - i \geq \quad & N - k + 1 \quad && \geq 1
\end{aligned}
$$

Let $x = N - k + 1$. Then (b) is equivalent to $[x, j], 1 \leq x \leq N - i$ which is (C) and therefore (b) is (C).

Similarly, we can show that (c) is equal to (B).

$$
\begin{aligned}
1 \leq \quad & k \quad && \leq i - 1\\
-1 \geq \quad & -k \quad && \geq -i + 1\\
N - 1 \geq \quad & N - k \quad && \geq N - i + 1\\
N \geq \quad & N - k + 1 \quad && \geq N - i + 2
\end{aligned}
$$

Let $x = N - k + 1$. Then (c) is equivalent to $[x, j], N - i + 2 \leq x \leq N$ which is (B) and therefore (c) is (B).

We can show that (d) is equal to (G) by first examining the bounds on the first component of (d), namely $\alpha = N - k + 1$.

$$
\begin{aligned}
i + 1 \leq \quad & k \quad && \leq N\\
-i - 1 \geq \quad & -k \quad && \geq -N\\
N - i - 1 \geq \quad & N - k \quad && \geq 0\\
N - i \geq \quad & N - k + 1 \quad && \geq 1\\
N - i \geq \quad & \alpha \quad && \geq 1
\end{aligned}
$$

The second component of (d) is $\beta = j - k + i$, which by substituting k is equivalent to $\beta = j - N + \alpha - 1 + i$. Therefore (d) is equivalent to $[\alpha, \beta] : 1 \leq \alpha \leq N - i, \beta = j - N + \alpha + i - 1$.

Now let us look at (G); its second component is $\gamma = j + k - (N - i + 1) = j - N + k + i - 1$. Therefore (G) is equivalent to: $[k, \gamma] : 1 \leq k \leq N - i, \gamma = j - N + k + i - 1$ which means

(d) is equivalent to (G). Similar reasoning shows that (e) is (F), (f) is (E) and (g) is (D). Since each part of the LHS matches a distinct part of the RHS, we have that LHS = RHS. Therefore, $\sigma([i,j]) = [N-i+1, j]$ is indeed a symmetry of N-queens.

This process can be automated for some cases. In the example shown here, the reasoning requires only syntactic manipulation and some basic arithmetic. However, this case has only binary constraints involving simple arithmetic expressions. It remains to be seen how far this method can be extended to other problems.

4.8.2 Proving via Constraint Program

Another approach to proving that a parametrised permutation is a symmetry is to use a constraint programming system. Here we describe how a constraint satisfaction program can be manually derived from a CSP model whose execution can prove or disprove a symmetry. Again, we illustrate this with the simple N-queens problem.

Similar to before, we have a simple representation of the problem:

```
forall i,j in 1..N where i \= j:
       q[i] != q[j]       /\
   q[i] - i != q[j] - j   /\
   q[i] + i != q[j] + j
```

The parametrised graph is shown below. Each literal in the graph is a pair $[i, v]$, representing $q_i = v$.

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

For the purposes of this example, we provide a simpler representation of the set of edges in the parametrised graph:

$$
\begin{aligned}
E[N] = \{([i,x],[j,y]) \mid\ & i \in 1..N, j \in 1..N, \\
& x \in 1..N, y \in 1..N, \\
& (i \neq j \wedge x \neq y) \vee \\
& (i \neq j \wedge x - i \neq y - j) \vee \\
& (i \neq j \wedge x + i \neq y + j)\}
\end{aligned}
$$

A parametrised permutation σ is a symmetry of the problem if, for a given edge e in the graph, $\sigma(e)$ is also in the graph. To extend this to model symmetries, the above must be true for all values of the model parameter N.

Formally, let E be the set of edges in the parametrised graph. We wish to show that $e \in E \Rightarrow \sigma(e) \in E$ for a given σ and for all e. In other words, we wish to prove that there exists no element such that $e \in E \wedge \sigma(e) \notin E$. We ask the constraint solver to find for us

```
symEdge(N,SI,SA,SJ,SB) :-
        integers([I,J,A,B]),
        % enforce bounds on literals, and ensure i != j
        % (1)
        1 $=< I, I $=< N,
        1 $=< J, J $=< N,
        1 $=< A, A $=< N,
        1 $=< B, B $=< N,
        ( I+1 $=< J ; J+1 $=< I ),

        % ensure that one of the constraints is violated.
        % (2)
        constraint(N,I,A,J,B),

        % relate Sx with x via the symmetry.
        % (3)
        SI $= N-I+1,
        SJ $= N-J+1,
        SA $= A,
        SB $= B.

constraint(N,I,A,J,B) :- A $= B.
constraint(N,I,A,J,B) :- I+A $= J+B.
constraint(N,I,A,J,B) :- I-A $= J-B.
```

Figure 4.5: Program to find a symmetric edge.

such a counterexample e – if the solver can prove that no such e exists, then the symmetry must hold.

Figure 4.5 shows an ECL^{i}PS^e program that uses an arbitrary constraint solver via `eplex` to find some $\sigma(e)$ where $e \in E$. First we ask the program to find some $e \in E$. An edge between the literals $q_i = a$ and $q_j = b$ has four parts: i, a, j, b. Each part must lie within its bounds (1), and the pair of literals must be disallowed by at least one of the constraints (2). Then we find the symmetric image of that edge (3). As before, the symmetry is a variable symmetry that reflects the chessboard, $\sigma([i, j]) = [N - i + 1, j]$.

If the *symEdge* predicate succeeds, the variables `SI,SA,SJ,SB` represent an edge $\sigma(e) = ([si, sa], [sj, sb])$ where it is known that $e \in E$. The next step is to try to show that $\sigma(e) \notin E$. If this can be proved, then the symmetry does not hold and e is a counterexample; however, if the solver exhausts the search space and fails, then no counterexample exists and the symmetry holds.

Figure 4.6 shows the ECL^{i}PS^e program that succeeds if the edge e, represented by its four components as above, is not in the set of edges E. An edge e is not in E if it lies outside the bounds (1) or if it satisfies every constraint (2). After a symmetric edge is found by *symEdge*, it is passed to *notEdge* to check whether it is truly in the set of edges.

We have tried this method with the four symmetries of N-queens that it can represent: the identity symmetry, the variable reflection, the value reflection and the composition of the latter two. In each of these cases, the solver fails to find a counterexample and proves that the symmetry holds. The only bound given by the user for N is that it is positive.

```
notEdge(N,I,A,J,B) :-
        % If any of these succeed, the element is outside the
        % bounds.
        % (1)
        (1 $>= I+1 ; I $>= N+1 ;
         1 $>= J+1 ; J $>= N+1 ;
         1 $>= A+1 ; A $>= N+1 ;
         1 $>= B+1 ; B $>= N+1) ;

         % If all of these succeed, the element satisfies all
         % constraints and therefore is in the set of edges.
         % (2)
        ((     A $>= B+1 ; B $>= A+1 ),
         ( A+I $>= B+J+1 ; A+I+1 $=< B+J ),
         ( A-I $>= B-J+1 ; A-I+1 $=< B-J )).
```

Figure 4.6: Program to determine whether an edge is in the graph.

In addition, we have tried to disprove permutations that are not symmetries of the problem. For example, a symmetry that arbitrarily interchanges the variables (or values) of the problem. In this case, the solver finds a counterexample edge between the literals $q_1 = 3$ and $q_3 = 1$ whose symmetric image is the edge between $q_1 = 3$ and $q_2 = 1$ under the symmetry that swaps q_2 and q_3. The symmetric image is not a edge in the graph. It is interesting to see the solver finds a counterexample for $N = 3$ and not for $N = 1$ or $N = 2$ as N must be at least 3 for a counterexample to be found. A similar counterexample is found for the interchangeable value case.

Finally, if the diagonal constraints are removed from the program – changing the N-queens problem into a variant of the Latin square problem – the above interchangeable variable and interchangeable value symmetries are shown to hold.

As with the first proving method, it is yet unknown how this method can be extended to other problems. One limitation already encountered is the inability to represent variable-value symmetries. However, the use of a constraint solver in the proving program has a benefit: any constraint that might appear in the CSP model can also be used in the proving program as is. It is possible that this approach can be automated, although that remains future work.

4.9 A General Framework for Detecting Properties

The framework presented in this chapter for detecting symmetries in constraint satisfaction models can be generalised to other properties. The steps performed by the framework can be tersely summarised as follows:

1. Find symmetries of several different instances of the model,
2. Generalise these instance symmetries to model symmetries,
3. Keep only those symmetries likely to apply to the model, and
4. Confirm that the symmetries do apply to the model.

There is nothing about this approach that is particular to symmetries. If we were to replace the symmetry-specific parts with more general terms, the approach would be:

1. Find properties that apply to instances of the model,
2. Generalise these instance properties to properties of the model,
3. Keep only those properties likely to apply to the model, and
4. Confirm that the properties do apply to the model.

For different properties the individual steps might vary greatly. The framework is in the same spirit as that used for generating implied constraints (Charnley et al., 2006), where the solutions of small instances of a constraint program model are manipulated to find additional logically true statements. In this section we discuss one potential application: finding opportunities for caching during search in a constraint program. As in symmetry detection, the knowledge of caching opportunities allows a search to exclude regions of the search space without missing solutions. However, unlike symmetries, determining where caching might be profitable is a difficult problem for which a practical automatic method has never been achieved. An automatic method that could find caching opportunities for an entire class of problems would be a very useful tool for improving the time required to solve many combinatorial problems.

In the remainder of the section we outline very briefly how our framework for symmetry detection could be generalised to find caching opportunities in constraint program models.

Caching is a modification to a search algorithm where the results of exploring some subtrees are stored and reused wherever possible. A subtree t does not need to be explored if a previously explored subtree p is known to contain all of the information that might be found in t; that is, any solution found in t is equivalent to, or worse than, a solution in p. Caching is only profitable, of course, where the number of occurrences of redundant subtrees is enough that it offsets the expense in time and space of storing results and detecting the redundancy of new subtrees to those already explored.

Example 49. Consider the CSP (X, D, C) where $X = \{a, b, c, d\}$, $D(a) = 1..3, D(b) = 1..4, D(c) = 1..5, D(d) = 1..6$ and the only constraint is $a + b + c + d = 10$ (see Figure 4.7). The search may proceed by asserting $a = 1$ and $b = 4$, leading to a subtree T_1. In this subtree there is the subproblem induced by the earlier assignments where $c + d$ must be made to equal 5. Later in the search, the initial decisions may be undone and instead the search may try $a = 2$ and $b = 3$, leading to a subtree T_2. The two subtrees T_1 and T_2 are equivalent, because every assignment over the variables $\{c, d\}$ that leads to a solution in T_1 also leads to a solution in T_2 (and vice versa). The two induced subproblems in T_1 and T_2 are the same.

□

Caching has proved very effective in reducing search time for some problems. For the minimum number of open stacks problem, Garcia de la Banda and Stuckey (2007) developed a dynamic programming approach that proved faster than any other previous approaches to the problem. Smith (2005) also shows the gains achievable via caching in constraint programming for satisfaction and optimisation problems.

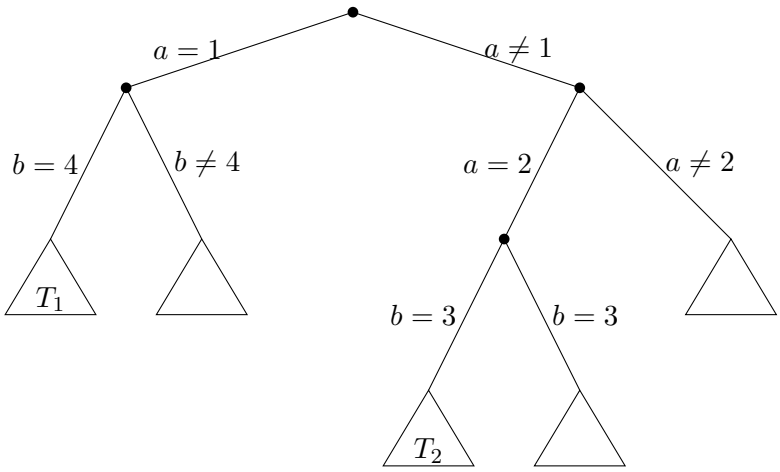


Figure 4.7: Part of a search tree showing equivalent subtrees T_1 and T_2.

Let us see how to apply the proposed framework to caching. The first step of the framework is to detect potential caching opportunities in individual instances. A simple but effective way to do this is to find all solutions of the instance and look in the resulting search tree for subtrees that are identical, as in the above example. Unfortunately, and unlike symmetries, caching opportunities may only exist under certain orderings of the variables during search. For this reason it is necessary to examine several search trees per instance, one for each variable order under consideration. Although finding all solutions of the problem and examining the search tree is expensive, the process needs to be carried out only for small instances.

The second step consists of the generalisation of caching opportunities to the problem class, which requires the notions of variable order and subproblem to be generalised. For example, a problem may have two sets X and Y of variables, where the size of the sets depends on the particular instance. A general variable ordering that is independent of instance is "label X first and then label Y"; a general subproblem is "the sum of Y must be less than some constant c". This step is the most difficult one. However, well-known variable orderings, such as labelling a matrix row-by-row or column-by-column, can be used as patterns to match against a concrete ordering for an instance. It is clear how such general notions can be instantiated to particular instances similar to how parametrised symmetries can be instantiated into instance symmetries.

This second step generalises the caching opportunities of instances into general patterns of caching opportunity which are conceptually similar to parametrised symmetry patterns. Because of this, the third step of filtering the potential caching opportunities can be done simply by taking the intersection of the patterns found for different instances.

An implementation of the framework for detecting caching opportunities could be used as a proposer of candidates for caching. In this form, the system would suggest potential caching opportunities and the user would exploit whichever ones they thought would be most useful, or easiest to use. The user would then be required to confirm that the proposed redundancy is correct, or subtrees may be incorrectly ignored. It is not necessary to confirm that the caching is actually profitable: the only damage that might be done is

that subtrees are stored where it will not be profitable but the search will find the same solutions as it would otherwise.

4.10 Conclusions

We have discussed in this chapter the automatic detection of symmetries in constraint program models. Unlike the symmetry detection method presented in the previous chapter, the approach used here can find symmetries in a whole class of problems rather than only in one instance at a time. This advantage greatly improves the practicality of the method, as symmetry detection does not need to be run on every instance to be solved or on large instances.

We have described a new framework that takes advantage of existing, and even future, powerful detection methods defined for problem instances by generalising their results to models without requiring these models to use any particular syntax or constraints. We have described a very simple, and incomplete, implementation that requires the problem to have matrix-like structure and only considers a pre-determined number of model symmetries: those that correspond to permutations of the objects in the matrix. While this is a very limited implementation of the general framework, it is nonetheless capable of detecting almost all of the symmetries in the benchmarks we have tested. Of course, more complete implementations of the framework will be able to detect even more kinds of symmetries.

While our approach has been shown to find almost all of the symmetries in our benchmarks, there remain some symmetries that it cannot find. The main source of incompleteness in our implementation is that only known symmetry patterns are found. However, it does find some of the most commonly occurring forms of symmetry, and it is these kinds of symmetry that we believe are the most profitable to exploit in symmetry breaking. This will be discussed in the next chapter. The approach is also limited to finding symmetries that appear in every instance of a CSP model. A symmetry that occurs due to the data and therefore appears only in some instances (such as where the data part is a graph that has a symmetry) cannot be found.

Automating the fourth step of the framework – proving that candidate symmetries are symmetries of the model – remains a challenging problem. There has been some previous work in the area, and we have demonstrated some simple approaches to tackle the problem here, but nevertheless there is still the lack of a reliable, easily used solution. It is fortunate that this lack is not a major barrier to the practical utility of the framework, but the approach would be much improved if this gap was to be filled. We are exploring other avenues to a solution to this problem, including automated theorem proving.

We have also discussed the generalisation of the framework to properties of constraint programs other than symmetries. The approach behind the framework is not specific to symmetry, and by replacing its components with others tailored to different properties, the framework can be used to detect other useful features of programs. One such example is caching, which like symmetry breaking is used to avoid searching redundant areas of the search space and reduce search time. Although this application has not yet been implemented, it shows the potential of this approach to be used for detecting properties of constraint programs.

In summary, our contribution in this chapter is a new framework that exploits the power of existing symmetry detection methods to find symmetries in CSP models. The major benefit of this approach is that symmetry detection needs to be performed only once per model, as the results apply to all instances of the model. In the next chapter, we tackle the next stage of handling symmetry in constraint programming: the use of the symmetries to improve the performance of searching for solutions.

Chapter 5

Symmetry Breaking

5.1 Introduction

To this point we have mainly discussed methods for finding symmetries in a given problem, where the problem is stated either as a CSP instance or as a CSP model. In this chapter we will discuss how the symmetries of a problem can be used.

The primary use of symmetries in constraint programming is for reducing the amount of effort required to solve problems. This is possible because symmetries cause redundancy. In particular, a symmetry causes a CSP to have redundant subtrees in its search space. A subtree of the search space is redundant if it can be obtained from another subtree simply by application of a symmetry, i.e. if it is symmetric to the other subtree. Redundant subtrees need not be explored: if subtree A and subtree B are symmetric, then any solution to the CSP that is contained in a subtree A has a symmetric counterpart in subtree B, and if subtree A has no solutions then subtree B has no solutions; therefore, only one of the subtrees A and B needs to be explored. By knowing what symmetries are present in a problem, one can determine which subtrees are redundant and avoid exploring them.

In this chapter we discuss *symmetry breaking*. Symmetry breaking is the term used to describe methods for reducing search effort by avoiding the exploration of the subtrees in the search space that are redundant due to symmetry. We explore some existing symmetry breaking methods and present a new method whose focus is on speed and practical utility. This new method, coined *Lightweight Dynamic Symmetry Breaking* (or *LDSB*), is an extension of the shortcut Symmetry Breaking During Search (SBDS) method (Gent and Smith, 2000) and fills a void in previous research by its being a viable generic symmetry breaking method, able to be used easily and effectively for many kinds of problems. LDSB focuses on symmetries that occur frequently in practice and that can be represented and manipulated efficiently. Interestingly, these symmetries include those detected by our symmetry detection method described in Chapter 4.

Our contributions are as follows. We identify common symmetry patterns that can be broken efficiently, formally define LDSB as an instance of the shortcut SBDS method for these symmetries, and prove that LDSB is correct and sometimes complete. Also, we show how LDSB is extended to combine symmetries. We discuss the techniques used by our ECL^{i}PS^e and Gecode implementations, techniques that can be incorporated into any

other constraint system. Finally, our experimental results show extremely good performance for our Gecode implementation, and very good comparative results for our ECL^{i}PS^e implementation: it significantly reduces the overhead required to exploit symmetries when compared to other dynamic symmetry breaking methods, and does not significantly increase the overhead when compared to a method without symmetry breaking. In fact, the results are so promising that we believe LDSB is general and practical enough to be used as a default dynamic symmetry breaking method. Parts of this chapter have previously been published in (Mears et al., 2008a).

5.2 Background

The purpose of symmetry breaking is two-fold. In some cases, one may wish to find a set of solutions where no two solutions are symmetrically equivalent. In other cases, one may tolerate symmetrically equivalent solutions and employ symmetry breaking simply to decrease the time to find one or many solutions to the problem. Let us introduce some concepts that will help us to illustrate these aims.

Definition 6. Let $S(P)$ be the set of symmetries of a CSP P. Two solutions A and B of P are *symmetrically equivalent* if there exists a symmetry $\sigma \in S(P)$ such that $\sigma(A) = B$.

Note that a solution is always symmetrically equivalent to itself regardless of its CSP, since the identity symmetry is present in every CSP.

A search for the solutions of a CSP P using symmetry breaking will find a subset T of the solutions of P. The set of solutions found is called a set of unique solutions if no two distinct elements of the set are symmetrically equivalent.

Definition 7. Let $\mathit{Sols}(P)$ be the set of solutions of a CSP P. A set $T \subseteq \mathit{Sols}(P)$ is a set of *unique solutions* of P if for every pair of solutions $A, B \in T$, $A \neq B$, A and B are not symmetrically equivalent, i.e. there does not exist any $\sigma \in S(P)$ such that $\sigma(A) = B$.

It is desirable for a symmetry breaking method to eliminate as much search as possible, but it should find at least one representative of each solution.

Definition 8. A symmetry breaking method is *correct* if it finds a set $T \subseteq \mathit{Sols}(P)$ of solutions of a CSP P and for every solution $A \in \mathit{Sols}(P)$ there exists a solution $B \in T$ and a symmetry $\sigma \in S(P)$ such that $\sigma(A) = B$. We say that the solution B is a *representative* of A.

Note that a solution may be a representative of many solutions.

Definition 9. A symmetry breaking method is *complete* if it is correct and it always finds a set of unique solutions.

Note that correctness of a symmetry breaking method is a strictly weaker condition than completeness: any symmetry breaking method that is complete must be correct, but a symmetry breaking method may be correct without being complete. For example, a symmetry breaking method that performs no symmetry breaking at all is correct but not

complete as it will find the entire set of solutions of a problem. If a CSP has no solutions at all, then any symmetry breaking method is complete for that problem.

In the rest of this section we will discuss the ways in which the symmetry of a problem can be used to improve the performance of search.

5.2.1 Reformulation

In some cases it is possible to transform a model that contains symmetries into an equivalent model that has fewer symmetries or no symmetries at all. In general, a given problem can be represented in many different ways and it is usually not possible to predict with certainty which one will be the most efficient.

Example 50. Consider the simple CSP $(\{x_1, x_2, x_3\}, 1..10, \{\mathit{all\_different}(\{x_1, x_2, x_3\}) \wedge \sum_i x_i = 15\})$. A solution of this problem is a triple of distinct integers whose sum is 15. There is a variable symmetry in this model: the order of the variables is irrelevant and x_1 can be swapped with x_2, x_2 with x_3 or x_1 with x_3. The symmetries will give rise to equivalent subtrees in the search tree whose exploration would be redundant. Also, multiple equivalent solutions would be found, such as $(2, 3, 10)$ and $(3, 10, 2)$. The problem can be remodelled using a set variable in the place of the three integer variables: $(\{s\}, \wp(\{1..10\}), \{\sum s = 15,\ \mathit{card}(s) = 3\})$. In this model, there is one set variable whose values must sum to 15. See Figure 5.1 for some solutions of the two problems. Clearly, the second formulation does not contain the variable symmetry of the first model, even thought the two models are equivalent; i.e. every unique solution to the first model has a corresponding solution in the second model and vice versa.

Unfortunately, even though a symmetry has been eliminated, it is not clear that this model's efficiency is any better than that of the first one. Whether it can be solved more quickly depends on the constraint solver's handling of integer and set constraints.

Integer model			Set model
x_1	x_2	x_3	s
1	4	10	$\{1, 4, 10\}$
6	7	2	$\{2, 6, 7\}$
8	4	3	$\{3, 4, 8\}$
4	1	10	$\{1, 4, 10\}$

Figure 5.1: Equivalent solutions of two different models for the same problem.

□

Despite this short-coming of reformulation, it has been applied successfully for some problems. For Kirkman's schoolgirl problem (a specialisation of the social golfers problem of Section A.2), Smith (2001) found a new representation that reduces the amount of symmetry and, coupled with a symmetry breaking method, improved search time. Likewise, Gent et al. (2005) gave a new model for the all-interval problem (problem 7 in CSPLib (Gent and Walsh, 1999)) that removes all symmetry and leads to dramatically improved performance.

Unfortunately, both of these improved models were derived for their specific problems. There is no known truly general method that we know of for reformulating models to make them more efficient, or less symmetric. Typically, finding a better model requires some insight into the problem itself.

5.2.2 Static Symmetry Breaking

One can reduce the amount of symmetries present in a CSP model by adding, before starting the search, constraints that prevent some, or all, of the redundant regions of the search space from being explored. This kind of method is called a *static* symmetry breaking method, following the terminology used in program analysis, because it is applied before run-time.

Example 51. □

Consider again the CSP $(\{x_1, x_2, x_3\}, 1..10, \{all\_different(\{x_1, x_2, x_3\}) \wedge \sum_i x_i = 15\})$ discussed in Example 50. A complete symmetry breaking method can be obtained by adding the ordering constraints $\{\mathbf{x_1} \leq \mathbf{x_2}, \mathbf{x_2} \leq \mathbf{x_3}\}$. These two less-than-or-equal constraints alter the CSP in such a way that the set of solutions to the altered CSP is a complete set of unique solutions to the original CSP. The ordering constraints ensure that only one representative of each solution will be found. For instance, in the original problem, each of the solutions $\{(2,3,10),(2,10,3),(3,2,10),(3,10,2),(10,2,3),(10,3,2)\}$ is a representative of the set but only the first of these six is permitted in the altered CSP.

It is always possible to add constraints to any CSP P to produce an altered CSP P' such that the solutions of P' form a complete set of unique solutions of P (Puget, 1993). Crawford et al. (1996) presented a method to generate constraints that when added to a model break variable symmetries. This requires adding one lexicographical constraint for each variable symmetry of the CSP. This method has a major shortcoming in that it requires one constraint per symmetry, rendering it infeasible when there are many symmetries.

A set of lexicographical constraints for symmetry breaking can in some cases be simplified without reducing their power. (Luks and Roy, 2002)

Example 52. Consider a CSP with a matrix of variables:

$$\begin{matrix} x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 \end{matrix}$$

Assume that the rows are interchangeable; i.e. $\langle x_1, x_2, x_3\rangle \leftrightarrow \langle x_4, x_5, x_6\rangle$. This symmetry can be broken by asserting that $\langle x_1, x_2, x_3\rangle$ is lexicographically less than $\langle x_4, x_5, x_6\rangle$. However, if it is known from the constraints of the problem that $x_1 \neq x_4$, the same effect can be obtained by the simpler constraint $x_1 < x_4$. □

In particular, if there is a variable symmetry over variables that are constrained to be different, the symmetry can be broken by a linear number of binary constraints instead of the exponential number of constraints needed for the general case (Puget, 2005b). Such

an improvement is not possible in general, but in some cases subsets of the full set of constraints can be used, resulting in incomplete symmetry breaking but still acceptable performance. Some research has focused on the special case of row and column permutations in matrix models (e.g. (Frisch et al., 2003a)), as these symmetries occur frequently in practice. Flener et al. (2002a) showed how a subset of the symmetries can be broken by lexicographically ordering the rows and columns of the matrix and that this method is effective in experiments. Another way to reduce the number of symmetry breaking constraints required is to combine the approach with computational group theory (CGT); Jefferson et al. (2006) showed that effect of lexicographical constraints can be simulated at less expense by the use of the group theory system GAP (Group, 2006).

A fundamental problem with using lexicographical symmetry breaking constraints is that it results in one solution being nominated in advance as the representative of a set of symmetrically equivalent solutions; i.e. that solution is the one to be found by the search and the others are excluded. This solution is not necessarily the solution that would have been found first in an unaltered search.

To illustrate this point, let us consider a search tree where the leaves of the tree are either failed nodes or solutions. The search finds the solutions from left to right in the tree; let us call the solutions in order A, B, C, etc. Let us assume that A, B and C are symmetrically equivalent. A symmetry breaking constraint may select C as the representative to be found and exclude A and B. However, the search may require a great deal of search before eventually exploring the path that leads to solution C, where without the symmetry breaking constraint the solution A may have been found with little search.

In some cases it is possible to choose the symmetry breaking constraints in such a way that they cooperate with the search strategies, this prevents the use of dynamic heuristics. In another special case of piecewise symmetry, Heller et al. (2008) showed that the drawback of static symmetry breaking where the variable ordering heuristic conflicts with the added constraints can be partly overcome by randomising the heuristics and the symmetry breaking constraints and restarting the search if it appears to be taking too long.

In summary, static symmetry breaking methods can be used to greatly reduce search effort without introducing significant overheads, thus yielding considerable speedups (see e.g. (Flener et al., 2002a; Puget, 1993)). However, the effectiveness of the methods usually relies on problem-specific solutions, and can vary dramatically due to conflicts with the search heuristics.

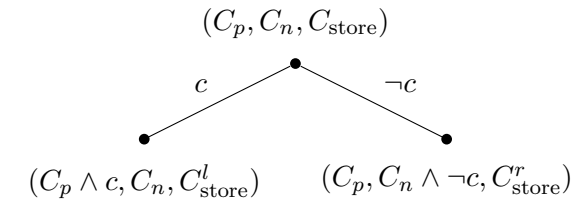
5.2.3 Dynamic Symmetry Breaking

An alternative approach to symmetry breaking is to inform the search algorithm of a problem's symmetries so that it can alter its exploration of the search tree to exclude symmetric regions. Such a method is called a *dynamic* symmetry breaking method, in contrast to a static symmetry breaking method. In this section we will look at some of the most commonly used dynamic symmetry breaking methods.

$\mathcal{S}$-*excluding* search tree

The $\mathcal{S}$-*excluding* search tree (Backofen and Will, 1999) posts additional constraints on the right-hand branches of decision nodes during search. That is, once a subtree of a node has been fully explored – on the node's left branch – it adds a constraint on the right branch to exclude any other subtrees that are symmetric to that one just explored.

The general case of a decision node in an $\mathcal{S}$-*excluding* search tree is shown in Figure 5.2. Given a set $\mathcal{S}$ of symmetries, an $\mathcal{S}$-*excluding* search tree is a finite, binary, rooted and ordered tree where every left edge is labelled by a constraint c, every corresponding right edge is labelled by $\neg c$, and every node v is labelled by the triple of constraints (C_p, C_n, C_{store}), where C_p and C_n are, respectively, the conjunction of left-branching (positive) and right-branching (negative) constraints on the path from the root to v, and C_{store} is the set of constraints in the CSP plus those dynamically added to exclude symmetric solutions. Note that this structure of the search tree does not restrict the order in which variables or values are explored, nor requires all values of a variable to be tried consecutively. These dynamically added constraints are computed as follows. Suppose the search is at some node $v = (C_p, C_n, C_{store})$ that branches left by adding constraint c and right by adding $\neg c$. Once the left subtree is fully explored, we can exclude anything symmetric to c in the subtree rooted at v. For a symmetry s, let s_{con} be the result of extending s to act on constraints. For example, for a variable symmetry that maps variable x onto $f(x)$ we could define $s_{con}(x = v) = (s(x) = v)$ (see Section 6.3.1 for details). Then, on the right branch, we would like to assert $\neg s_{con}(c)$ for every symmetry s that is *active* at v, i.e., for every symmetry for which $s_{con}(C_p)$ holds. This is achieved by posting $s_{con}(C_p) \Rightarrow \neg s_{con}(c)$ on the right branch. This constraint can be read as: if the image of the constraints C_p under s_{con} holds, then the negation of the image of the branching constraint c under s_{con} can be posted.



$$C^l_{\text{store}} = P(C_{\text{store}} \wedge c)$$
$$C^r_{\text{store}} = P(C_{\text{store}} \wedge \neg c \wedge \bigwedge_{s \in \mathcal{S}} s_{con}(C_p) \Rightarrow \neg s_{con}(c))$$

Figure 5.2: A node in an $\mathcal{S}$-*excluding* search tree (by Backofen and Will (1999)).

This theoretical method is very general and has been proved correct and complete, but has never been implemented in its general form. In fact, as pointed out in Chapter 6, it is not clear how s_{con} is defined for many kinds of constraints. An instance of the method for the particular case of $x = v$ branching constraints has independently been defined and implemented by (Gent and Smith, 2000). This instance is discussed in the next section.

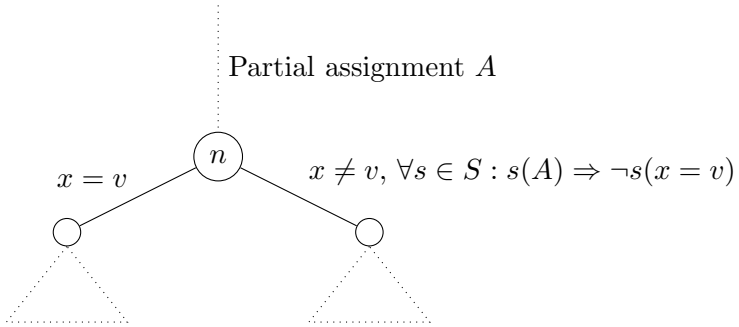


Figure 5.3: Overview of SBDS decision point.

Symmetry Breaking During Search

The *Symmetry Breaking During Search*, or *SBDS* method can be seen as an instance of the $\mathcal{S}$-*excluding* search tree but was independently developed by Gent and Smith (2000). The core idea of SBDS is the same as for the $\mathcal{S}$-*excluding* search tree: once a region of the search tree has been explored then, upon backtracking, any other symmetric region of the search tree can be excluded.

Assume we have a CSP $P = (X, D, C)$ that has the group of symmetries S. SBDS is based on a depth-first search of a tree (see Figure 5.3) whose nodes have zero or two children: the left one (labelled $x = v$) binds variable x to value v, while the right one (labelled $x \neq v$) eliminates v from the domain of x, where $x \in X$ and $v \in d(X)$. As for the $\mathcal{S}$-*excluding* search tree, this structure of the search tree does not restrict the order in which variables or values are explored. Consider a node n in the search tree with two children as described above. Let the partial assignment leading to node n – the set of labels of the left branches on the path from the root to n – be called A.

The search always follows the left child of n first, asserting $x = v$ and exploring that region of the search tree. Once that region has been explored, the search backtracks to n and prepares to follow the right child. At this point, we know that every possible assignment that includes $A \cup \{(x = v)\}$ has been examined – this is exactly what is searched in the left subtree of n. Therefore, it is correct to exclude any assignment which is symmetric to $A \cup \{(x = v)\}$. SBDS excludes any such assignments by posting on the right child of n, for each symmetry $s \in S$, a constraint of the form:

$$A \wedge \neg(x = v) \wedge s(A) \Rightarrow \neg s(x = v)$$

If the constraint solver permits the posting of "local" constraints (which are enforced only in a certain region of the search tree) then the constraint can be simplified to:

$$s(A) \Rightarrow \neg s(x = v)$$

since the A and $\neg(x = v)$ components are guaranteed to hold in that subtree (see Figure 5.3).

SBDS has been proved correct and complete (Backofen and Will, 1999); however, it suffers from some practical issues. There are five main sources of potential inefficiency

in SBDS. The first is *vacuous* posting, i.e. posting constraint $s(A) \Rightarrow \neg s(x = v)$ when $s(x = v) = (x = v)$ and $s(A)$ holds, since this achieves no pruning ($\neg s(x = v)$ is equal to the already posted $\neg(x = v)$). The second is *multiple posting* due to symmetries that yield identical conditional constraints. The third is not eliminating *broken* symmetries, i.e. those for which condition $s(A)$ is inconsistent with A – these symmetries can never give any more pruning. The fourth is not posting a constraint that could have achieved pruning, for example by not considering every symmetry of the problem. The fifth and last is that if the group of symmetries is large, a serious burden is placed on the user, who has to provide a function describing the effect of each symmetry, and on the implementation, which must process each symmetry in the conditional constraints.

The GAP-SBDS (Gent et al., 2002) extension of SBDS avoids these issues by using the computational group theory system GAP to process the entire group of symmetries. However, while GAP can handle large groups efficiently, the extra prunings (if any) might not necessarily be enough to pay for its own significant cost. Furthermore, GAP is not available in many constraint systems. The fifth problem can be alleviated by using only a subset of the symmetries, as can be done for static symmetry breaking; for example, it was shown by McDonald and Smith (2002) that applying only a subset of the symmetries can improve search times. The main advantage of SBDS, and of all dynamic symmetry breaking methods, is that it cooperates fully with the search's variable and value ordering. Unlike static symmetry breaking methods, the first solution that would be found without symmetry breaking is always a representative chosen to be found by a dynamic symmetry breaking method.

Shortcut SBDS

In the form described above, SBDS requires one constraint for every symmetry to be broken. Unfortunately, and as mentioned before, even for small problems there may be very many symmetries. For example, the Latin square problem of size 5 already has 1728000 symmetries, and the number of symmetries grows exponentially as the size increases. To reduce the overhead of SBDS, Gent and Smith (2000) proposed a shortcut form of SBDS that posts fewer constraints. The "shortcut SBDS" methodology is based on the generic SBDS, but requires a problem-specific implementation that reduces the amount of work done.

Gent and Smith (2000) discussed the idea by means of the graph colouring problem. This problem can be modelled as a CSP (X, D, C) where there is one integer variable $x_i \in X$ for each vertex of the graph, $D = 1..n$ where n is the maximum permitted number of colours, and C enforces the condition that adjacent vertices (variables) must have different colours (values). Apart from symmetries that may occur in the graph itself, this CSP has interchangeable values because the colours are interchangeable. During the search for a solution to this problem, assume that a partial assignment A has been made and the search branches on $x_i = v$ and $x_i \neq v$. The only useful symmetries at this point – those that can yield any pruning of the search tree – are those symmetries s such that $s(A) = A$. These symmetries are exactly those that change none of the colours used in A. We can further reduce the set of useful symmetries by excluding any symmetry s where $s(x \neq v) = (x \neq v)$ – that is, any symmetry that eliminates already pruned values. Because s must not change a colour that has already been used, v must be an unused

colour, and its image under s must also be an unused colour. In short, the only time additional pruning can be performed is when a new colour is used, and if that fails, no other unused colour should be tried in its place.

This chain of reasoning leads to a simple rule for this example that is easy to implement. Unfortunately, it requires effort on behalf of the programmer to derive the reduced shortcut rule. Further, the reasoning that leads to a simple rule for this problem does not transfer to all problems, and the effort required to derive it must be then repeated for different problems. As will be seen later in the chapter, our new method of symmetry breaking is based on the idea of shortcut SBDS but relieves the programmer of this burden by automating the process for certain common kinds of symmetry.

Symmetry Breaking by Dominance Detection

An alternative to the SBDS method is *Symmetry Breaking by Dominance Detection*, or *SBDD* (Focacci and Milano, 2001; Fahle et al., 2001). Like SBDS, SBDD works by excluding redundant parts of the search space during the search, but its manner of doing so is quite different.

At each node in the search, SBDD asks: is this node symmetrically dominated by a node we have already seen? If the answer is yes, then the current node does not need to be explored and the search backtracks; if the answer is no, then search continues as normal. To fully understand how this works, we must understand the notion of *no-goods*. The following definitions are based on those by Puget (2002).

Definition 10. A node w is a *no-good* with respect to a node n if:

- the subtree rooted at w is explored before n, and
- w is not an ancestor of n.

Therefore, the no-goods of a node n are the roots of the subtrees that are completely explored before n is encountered.

Definition 11. Let n and w be nodes in a search tree, D be the set of branching decisions leading to node w, and A be the partial assignment at n. Then, n is *symmetrically dominated* by w if:

- w is a no-good with respect to n, and
- there exists a symmetry s such that $s(D)$ is entailed by A.

The intuition behind this definition is as follows. The subtree rooted at w – all of whose nodes are extensions of D – has been explored. By symmetry, anything that extends $s(D)$ for any symmetry s is symmetric to something that extends D and therefore need not be explored. If A entails $s(D)$ for some s, then n need not be explored. Any node that is dominated by another node need not be explored.

A search algorithm using SBDD performs a check at every node n of the search space to determine whether n is dominated. This check consists of determining the existence of a symmetry s and a no-good w such that $s(D) \dashv A$ as described above. The implementation of this check is key to the efficiency of SBDD. As for SBDS, SBDD cooperates well with variable and value orderings.

Example 53. Assume the search is at node n where the partial assignment is $A = \{x_1 = 2, x_2 = 1, x_3 = 3\}$ and there are variable symmetries among x_1, x_2, x_3 such that they are mutually interchangeable. One possible dominance check is to examine the values taken by x_1, x_2, x_3 – in this case $\{1, 2, 3\}$ and check if there exists a no-good w whose decisions D apply to a subset of $\{x_1, x_2, x_3\}$ and where the values taken by $\{x_1, x_2, x_3\}$ are a subset of those taken by the same variables in A. □

SBDD can be used for problems where the number of symmetries is too many for SBDS to handle efficiently. However, SBDD requires the dominance checking function to be tailored for the problem at hand, or else a generic dominance check to be used (as discussed in the next section). Using a problem-specific check imposes a burden on the user, while using a generic check may increase the overhead of the search as the possibly slow dominance check is performed at every search node.

Computational Group Theory and Dynamic Symmetry Breaking

Both SBDS and SBDD can benefit from being combined with a system for computational group theory. In SBDS, the group theory system aids in the representation and manipulation of a large number of symmetries: it becomes unnecessary to list every individual symmetry of the problem. Instead, a programmer can list only the set of generating symmetries. For SBDD, group theory has been used to create a generic dominance checker based only on the problem's group of symmetries, reducing the algorithmic effort required by the programmer. The two symmetry breaking methods, linked with the computational group theory package GAP (Group, 2006), are known as GAP-SBDS (Gent et al., 2002) and GAP-SBDD (Gent et al., 2003).

GAP-SBDS and GAP-SBDD have publicly available implementations in the ECL^{i}PS^e constraint programming platform. However, they are not without drawbacks. GAP-SBDS clearly raises the number of symmetries that can be efficiently handled with SBDS, taking it from the order of thousands into the billions. Unfortunately, even billions is not enough for highly symmetric problems where the size of the symmetry group grows exponentially (e.g. the BIBD problem tested by Gent et al. (2003)). GAP-SBDD has been tried in practice with very large groups, but has been shown in some cases to perform worse than GAP-SBDD due to the interaction between SBDD and propagation on variables that are not labelled by the search (Petrie and Smith, 2003; Petrie, 2003).

5.3 Lightweight Dynamic Symmetry Breaking

As discussed before, static symmetry breaking methods can be used to greatly reduce the search effort when solving problems with symmetry. In some cases, the symmetry breaking constraints can be simplified to give a set of simple constraints that can break most, or all, symmetry. However, in general it is necessary to use complex lexicographical ordering constraints that are more expensive to propagate. More importantly, the addition of constraints can alter the order in which solutions are found; i.e. the solution that would have been found first may be now excluded, and the search may have to waste time exploring areas that are now fruitless due to the symmetry breaking constraints. Whether this occurs can depend on the particular search heuristics used.

On the other hand, dynamic symmetry breaking methods do not suffer from this problem. Whichever solution would have been found first without symmetry breaking will still be the first solution found. However, dynamic methods require extensions to the search algorithm that can add considerable overhead to the search. The overhead comes from the need to keep track of which symmetries are still active as the search progresses, of from the need to check whether a given node is dominated. In general, to consider every symmetry of the problem requires a great deal of processing, or the use of a potentially time-consuming computational group theory system such as GAP.

Given this state of affairs, it is not clear to the uninitiated constraint modeller exactly how to make use of symmetry breaking for their program. The wrong choice of method may give worse performance than even the naive search; conversely, to avoid symmetry breaking altogether may be to miss an opportunity to drastically improve search time. Of course, one might try several symmetry breaking methods and compare their performance, but this can require significant effort – especially if problem-specific static constraints or dominance checks need to be provided – possibly for little gain.

This gap in constraint programming practice prompted our development of the *Lightweight Dynamic Symmetry Breaking* method, or *LDSB*, which can be seen as an extension of the shortcut SBDS method. The aim of LDSB is to provide a symmetry breaking method with the following properties.

- **Automation.** LDSB can be used in a constraint program by simply specifying a generating set of symmetries for the problem.
- **Efficiency.** LDSB often achieves good reductions in search time and only rarely yields significant slowdowns, relative to the same search without symmetry breaking.
- **Generality.** LDSB can be applied in conjunction with any variable or value ordering, and to a problem with any kind of constraints, whether the task is to find one or many solutions, and do so while maintaining efficiency and without the need to modify the problem.

The desired properties of LDSB have guided us in making decisions regarding the design of the algorithm. Our choices were also inspired by the shortcut SBDS method (Gent and Smith, 2000), which restricts a complete and generic method for the sake of efficiency. The decisions we made and their justifications are described as follows:

Dynamic Symmetry Breaking

LDSB performs symmetry breaking by altering the search algorithm, rather than altering the problem itself. This is because we believe it is easier to achieve our three aims using dynamic rather than static symmetry breaking; in particular, automation and generality are straightforward. The main challenge then is to increase efficiency by reducing overhead. Of course, this does not position our method as a substitute for static symmetry breaking. Instead, we believe that the two kinds of symmetry breaking are complementary and constraint platforms should have both. Which kind is used will depend on whether a small number of efficient constraints can be added to break the symmetries.

Focus on Common Symmetries

There are several patterns of symmetry that occur commonly in constraint programs, are easy to represent and whose exploitation yields large reductions in search space. For LDSB to be efficient it is vital that it performs well in the most common cases. This concern has led to the choice of a representation for symmetries in LDSB that allows the common kinds of symmetry to be expressed in a way that is efficient to represent and manipulate. The advantage of this approach is a significant reduction in overhead since LDSB does not require the expense of calling a separate general-purpose CGT package to examine the symmetry groups and instead does only the simplest kinds of group theory. Unfortunately, this comes at the expense of those symmetries that occur less frequently. In particular, LDSB is unable to handle variable-value symmetries.

Combination of Symmetries

Any symmetry can be composed with another symmetry (or itself) to give another possibly different symmetry. This raises a question for symmetry breaking methods: if the user has specified a set of symmetries, should the method try to break new symmetries formed by the composition of the given symmetries?

This an issue both of automation and efficiency. A symmetry breaking method that does not consider composed symmetries can be made to behave like one that does by requiring the user to explicitly specify the composed symmetries in addition to the original set. However, we believe this requirement imposes an excessive burden on the user, as the full set of symmetries may be very large. For example, the symmetries among n interchangeable variables can be derived by composing only two symmetries, while the full set has a total of $n!$ symmetries. LDSB handles composed symmetries without the user having to specify the compositions, both to ease the user's burden and to possibly reduce the search space without having to explicitly list every possible useful combination of symmetries.

Incompleteness

As discussed previously, symmetry breaking is often used to improve the time taken to find solutions to a problem, but may also be used to guarantee that only symmetrically unique solutions will be found. We believe that the former is the primary motivation for symmetry breaking and, as such, LDSB favours speed over completeness. As a result it is not essential for LDSB to exclude all symmetric regions of the search space; it may explore two subtrees that are symmetrically equivalent.

A benefit of this decision is that LDSB can use a simple representation for symmetries, one that does not need to keep track of every potential opportunity for reducing the search space. The simplicity of the representation leads to faster algorithms for processing the symmetries during search.

5.4 Common Symmetry Patterns

Certain patterns of symmetry occur frequently in constraint programs. This is particularly the case when the underlying problem has some structure – this was seen in Section 4.5.2, where common symmetry patterns are described in terms of a matrix-like formation. As described above, the efficient exploitation of these symmetries is crucial to the success of a general-purpose symmetry breaking method.

The following patterns of symmetry are represented in LDSB. We have identified them as occurring often in constraint programs, being amenable to efficient representation, and worthwhile to exploit during the search. In addition, these symmetries can be detected automatically in CSP models: each symmetry pattern maps to one or more of the parametrised patterns discussed in Section 4.5.2. The symmetry patterns are described in terms of a given CSP $P = (X, D, C)$.

5.4.1 Interchangeable Variables

A set of *interchangeable variables* is a set of variables $W \subseteq X$ whose members are interchangeable. That is, any permutation of the variables in W is a symmetry of the problem. This symmetry is an instance of the value swap pattern of Section 4.5.2, where the variable indices are the values being swapped.

Example 54. Consider a CSP (X, D, C) where all variables $x_i \in X$ have the same domain and the only constraint is $\sum_{x_i} x_i = a$ where a is a constant. All variables in X are interchangeable. □

5.4.2 Interchangeable Values

A set of *interchangeable values* is a set of values $V \subseteq D$ whose members are interchangeable. That is, any permutation of the values in V is a symmetry of the problem. This symmetry is also an instance of the value swap pattern of Section 4.5.2, where it is the value dimension whose values are swapped.

Example 55. Consider a task allocation problem where there is a set of n tasks, each to be executed by an individual from a group of m people, and constraints to specify which pairs of tasks cannot be performed by the same person. This problem can be modelled by a set of integer variables $x_1, x_2, \ldots, x_n$, whose domain is $1, 2, \ldots, m$ and where $x_i = j$ means that person j executes task i. The constraints are of the form $x_a \neq x_b$ for every pair of tasks (a, b) that must be performed by different people. In this problem the people are interchangeable, and therefore the values $1, 2, \ldots, m$ form a set of interchangeable values. □

5.4.3 Interchangeable Variable Sequences

A set of *interchangeable variable sequences* is a set of sequences of variables whose members are interchangeable. That is, for any two sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, the permutation $\sigma(a_i) = b_i$ is a variable symmetry. This symmetry can represent some instances of the value swap pattern, the dimension invert pattern and the dimension swap pattern described in Section 4.5.2.

Example 56. Recall the Latin square problem (see Section A.8) of size n whose aim is to fill an $n \times n$ matrix with values from 1 to n such that all of the values within a single row or column are different. A natural model for this problem is to have one integer variable whose domain is $1, 2, \ldots, n$ for each of the n^2 elements of the matrix, as in Example 40. Consider each row or each column of the matrix as a sequence of variables. Under this model, the set of rows is a set of interchangeable variable sequences and the set of columns is another set of interchangeable variable sequences.

Another symmetry of the Latin square problem is that which reflects the matrix around the diagonal, interchanging the variable at position (i, j) with the one at (j, i). This can also be seen as a set of interchangeable variable sequences with two elements. If x_{ij} is the variable at position (i, j), then the first sequence would be $(x_{12}, x_{13}, x_{23}, x_{14}, \ldots)$ and the second would be $(x_{21}, x_{31}, x_{32}, x_{41}, \ldots)$. □

5.4.4 Interchangeable Value Sequences

A set of *interchangeable value sequences* is a set of sequences of values whose members are interchangeable. That is, for any two sequences $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_n$, the permutation $\sigma(a_i) = b_i$ is a value symmetry. This symmetry can represent some instances of the value swap pattern and the dimension invert pattern described in Section 4.5.2.

Example 57. Recall the N-queens problem, whose aim is to place n queens on an $n \times n$ chessboard so that no queen attacks another. This problem can be modelled with one integer variable per queen, each with domain $1, 2, \ldots, n$. A value symmetry of this model replaces each value x with $n - x + 1$, representing a reflection of the chessboard. This symmetry can be seen as a two-element set of interchangeable value sequences: $(1, 2, \ldots, \lfloor \frac{n}{2} \rfloor)$ and $(\lceil \frac{n}{2} \rceil + 1, \lceil \frac{n}{2} \rceil + 2, \ldots, n)$. □

5.5 Symmetry Representation and Search

The search algorithm of LDSB is similar to that of SBDS: it is a depth-first exploration of the search space, where every branching point is a binary decision between $x = v$ and $x \neq v$ for some variable x and some value v in the domain of x. Like SBDS, LDSB also posts constraints on the $x \neq v$ branch to break symmetries. However, unlike SBDS and like the shortcut form of SBDS, not every symmetry is used to post constraints at every decision node. Only symmetries that are *active* will be used to calculate the additional symmetry breaking constraints. As in the shortcut SBDS, a symmetry s as active in the context of a partial assignment A if $s(A) = A$. Figure 5.4 shows an overview of a decision point in an LDSB search.

At each search node, the left branch is examined first. The branching constraint $x = v$ is posted, and each symmetry representation is updated, depending on its type as described below. Note that no pruning is done on the left branch. After the subtree under the left branch is completed, the right branch is tried. The branching constraint $x \neq v$ is posted, and symmetry breaking constraints are also computed and posted. Again, the details depend on the kind of symmetry.

We have implemented LDSB in the ECL^{i}PS^e and Gecode constraint programming platforms. The implementations are similar but vary in some specific points. In the remainder

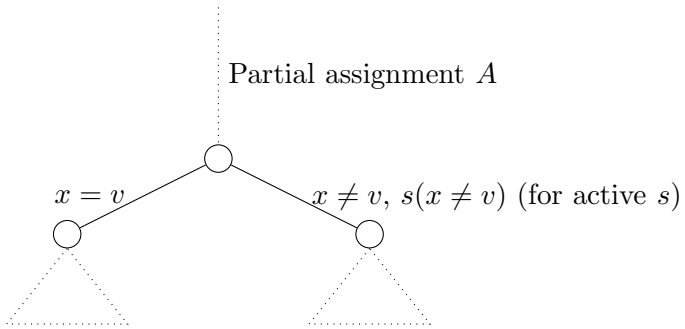


Figure 5.4: Overview of an LDSB decision point.

of this section, for each common symmetry pattern, we describe the representation of the symmetry in our two implementations and the algorithms to process it, and prove that the algorithm is correct (finds at least one representative for every solution).

5.5.1 Interchangeable Variables

Consider a CSP $P = (X, D, C)$, where $W \subseteq X$ is a set of interchangeable variables. Let r be any permutation of W, and let f_r denote both its induced solution symmetry and (by an abuse of notation) its associated solution-preserving function σ_{f_r}. Let the search be at node n with current assignment α, and let $G(W, n)$ be the group of permutations of W that are active at node n (i.e., any r for which $f_r(\alpha) = \alpha$). Let us also assume that none of the variables in W appears in α (i.e. $scope(\alpha) \cap W = \emptyset$). It is clear that $G(W, root) = G(W, n)$, i.e. all permutations of W are active at any node from the root to n. In the following we will describe how the program variable `ListW`, which represents the set W, changes during search.

Suppose now that the first child of n is labelled $x = v$ with associated assignment $\beta = \alpha \cup \{x = v\}$ where $x \in W$, i.e. x is the first variable in W to occur in β. Clearly, $f_r(\beta) \neq \beta$ for every r such that $r(x) \neq x$ and, therefore, every such f_r is broken (i.e., no longer active) at node $x = v$. As a result, `ListW` does not need to represent these broken symmetries any longer, and they are deleted simply by removing x from `ListW`. Note that f_r might become active again at some descendant n' of node $x = v$. This can occur if there is a set $V \subset W$, $x \in V$ with two or more variables and the assignment at n' contains $y = v : y \in V$. Despite this, eliminating f_r after node n is not a problem since any potentially lost pruning will be done by another symmetry; i.e. there must be another variable symmetry r' with $r'(y) = y : y \in V$ and $r'(z) = r(z) : z \notin V$ that has remained active throughout the path to n', is represented by the remaining list `ListW` $\setminus V$, and in the current path is broken by the same constraints as r.

Upon backtracking, the second child of node n will be labelled $x \neq v$. At this point, for any $r \in G(W, n)$ such that $f_r(\beta) \neq \beta$, we add $\neg r(x) = v$ and so doing prune every part of the subtree symmetric to β under f_r. This is exactly the same as adding `Y \= v` for every `Y` in `ListW` and it is correct since, for every such `Y`, the variable symmetry $f_r(x = v) = (y = v)$ is represented by `ListW`.

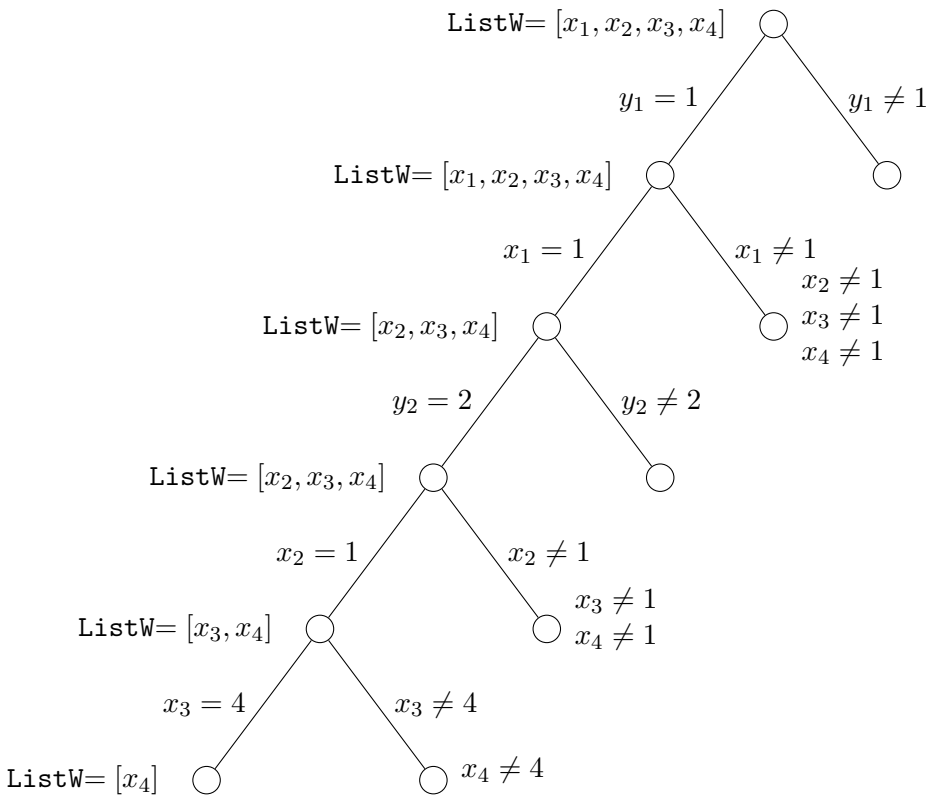


Figure 5.5: Evolution of `ListW` during search for interchangeable variables. Symmetry breaking constraints are shown to the right of $x \neq v$ nodes.

In summary, at every decision node n that branches on $x = v$: on the left branch, x is removed from `ListW` (if it is present); on the right branch, if x is in `ListW` then $y \neq v$ is posted for every y in `ListW`.

Example 58. Consider the CSP $(\{x_1, x_2, x_3, x_4, y_1, y_2\}, \{1, 2, 3\}, C)$ where there is the set of interchangeable variables $W = \{x_1, x_2, x_3, x_4\}$. Figure 5.5 shows the effect on `ListW` of each particular binding. Importantly, `ListW` only changes whenever the binding involves a variable in `ListW`. Also, note that once $x_1 = 1$ is posted, the variable symmetry $r \equiv < x_1, x_3 > \leftrightarrow < x_2, x_4 >$ is not active, since it maps x_1 to x_2; that is why it is not represented by `ListW` at node $x_1 = 1$. However, r becomes again active at node $x_2 = 1$. The fact that `ListW` does not represent r at node $x_1 = 1$ is however not a problem, since it does represent the symmetry $< x_3 > \leftrightarrow < x_4 >$ which will achieve as much pruning as r once both x_1 and x_2 have the same value. □

LDSB is correct for a set of interchangeable variables. To prove that LDSB breaks interchangeable variable symmetries correctly, we rely on the correctness of an $\mathcal{S}$-*excluding* search tree, shown by Backofen and Will (1999). We show merely that LDSB does a subset of the pruning done by an $\mathcal{S}$-*excluding* search tree and is therefore correct. Note that all

extra pruning added by LDSB occurs on the right branch, where $x \in W$. Therefore, we need only to focus on this branch.

Lemma 7. For the CSP (X, D, C) with the set of interchangeable variables $W \subseteq X$, and at a node which branches on $x = v$ and $x \neq v$ where $x \in W$, LDSB prunes on the right branch a subset of the prunings done in an $s - excluding$ search tree.

Proof. First, let $W' = W \setminus L$, where L is the set of variables that have been used in a left branch on the path from the root to the current branching node. By definition, x is in either L or W' – it cannot be in neither or both. If $x \in L$, then x already has a value and the search would not branch on $x = a$; therefore, $x \in W'$.

Let G be the group of symmetries due to the interchangeable variables W. An $s - excluding$ search tree posts on the right branch $s(C_p) \rightarrow \neg s(x = v)$ for each symmetry $s \in G$, where C_p is the set of left branch constraints posted on the path from the root to current branching node. We assume that every left branch is of the form $Var = Val$ and therefore C_p is a set of literals.

LDSB posts $\neg(w = v)$ for all $w \in W'$. To show that the pruning done by LDSB is a subset of an $s - excluding$ search tree's, we must show that for each w there exists a symmetry $s \in G$ for which $s(C_p)$ holds and $s(x) = w$. To do so, for any given $w \in W'$, we construct a symmetry s such that $s(x) = w$ and show both that $s(C_p)$ holds and that $s \in G$. First, we define $s(x) = w$, $s(w) = x$ and $s(y) = y$ for all $y \in X \setminus \{x, w\}$. It can be seen that neither x nor w is involved in any literal in C_p: if x is involved in a literal of C_p, then $x \in L$ and therefore $x \notin W'$, which contradicts the assumption made earlier; similarly, if w is involved in C_p, then $w \notin W'$ which contradicts the definition of w. Since neither x nor w is involved in C_p, then $s(C_p) = C_p$. Since C_p is known to hold, $s(C_p)$ also holds.

The remaining step is to show that $s \in G$. Since G contains every permutation of the variables of W, it must include s (which interchanges only x and w) because x and w both are in W. □

LDSB's symmetry breaking for interchangeable variables is optimally efficient, i.e. it does not post vacuous constraints, does not perform multiple postings, eliminates all broken symmetries, and is complete. Thus, it builds the same search tree as SBDS for a single set of interchangeable variables.

Lemma 8. A search tree produced by LDSB for CSP (X, D, C) with interchangeable variables W has no distinct symmetric nodes. That is, for any two distinct nodes n_α and n_β in the search tree, there is no symmetry $\sigma_r \in G(W)$ such that $\sigma_r(\alpha) = \beta$.

Proof. We will prove the above proposition by contradiction. Assume the existence of two distinct nodes in the search tree, n_α and n_β, and some $\sigma_r \in G(W)$ such that $\sigma_r(\alpha) = \beta$. We will show that such an arrangement cannot occur.

Let n_γ be the lowest common ancestor of n_α and n_β, and let us denote its children as $n_{x=v}$ and $n_{x\neq v}$. Without loss of generality, let n_α be a descendant of $n_{x=v}$ (or that node itself) and let n_β be a descendant of $n_{x\neq v}$ (or that node itself). Therefore, $x = v \in \alpha$, $x \neq v \in \beta$ and $r(x) = v \in \beta$.

It can be seen that $x \neq r(x)$, because if $x = r(x)$ then β must contain both $x = v$ and $x \neq v$, which is impossible. Since $r(x) \neq x$, then x must be in W. Furthermore, x must be in W' because it is in W and has not yet been given a value by the search.

Let us denote $r(x)$ as y and first consider the case where $y \in W'$. Let $P(W)$ be the set of permutations of W. Since x and y are both in W', and by definition LDSB asserts $r(x) \neq v$ at $n_{x \neq v}$ for all $r \in P(W')$, $y = v$ will be pruned at node $n_{x \neq v}$ and therefore n_β cannot be reached. This means that y cannot be in W'.

Let us now consider the case where $y \notin W'$. Consider the sequence $[x_1, x_2, \ldots, x_n]$ where $x_i = r^i(x)$ and n is the smallest positive integer such that $r^n(x) = x$. Note that $y = x_1$ and $x = x_n$. There exists some smallest $1 \leq i \leq n-1$ such that $x_i \notin W'$ and $x_{i+1} \in W'$.

The value given to y must be v, because $y = v \in \beta$. If $x_2 \notin W'$, then x_2 must also have the value v, because $x_2 = v \in \beta$. By induction, $1 \leq j \leq i$, x_j has the value v. Therefore, x_i must have the value v and $x_{i+1} = v \in \beta$. Because both x and x_{i+1} are in W', LDSB will assert $x_{i+1} \neq v$ at $n_{x \neq v}$ as before. Clearly, this is impossible.

Therefore, y cannot be in W' and y cannot be not in W'; so no such tree exists. □

5.5.2 Interchangeable Values

Interchangeable values are handled very similarly to interchangeable variables. Consider a CSP $P = (X, D, C)$, where $W \subseteq \bigcup_{d \in D} d$ is a set of interchangeable values. Let r be any permutation of W, and let f_r denote both its induced solution symmetry and (by an abuse of notation) its associated solution-preserving function σ_{f_r}. Let the search be at node n with current assignment α, and let $G(W, n)$ be the group of permutations of W that are active at node n (i.e., any r for which $f_r(\alpha) = \alpha$)). Let us also assume that none of the values in W appears in α. It is clear that $G(W, \mathit{root}) = G(W, n)$, i.e. all permutations of W are active at any node from the root to n. In the following we will describe how the program variable `ListW`, which represents the set W, changes during search.

Suppose now that the first child of n is labelled $x = v$ with associated assignment $\beta = \alpha \cup \{x = v\}$ where $v \in W$, i.e. v is the first value in W to occur in β. Clearly, $f_r(\beta) \neq \beta$ for every r such that $r(v) \neq v$ and, therefore, every such f_r is broken (i.e., no longer active) at node $x = v$. As a result, `ListW` does not need to represent these broken symmetries any longer, and they can be deleted simply by removing v from `ListW`.

Upon backtracking, the second child of node n will be labelled $x \neq v$. At this point, for any $r \in G(W, n)$ such that $f_r(\beta) \neq \beta$, we add $\neg x = r(v)$ and in so doing prune every part of the subtree symmetric to β under f_r. This is exactly the same as adding `X \= w` for every `w` in `ListW` and it is correct since, for every such `w`, the variable symmetry $f_r(x = v) = (x = w)$ is represented by `ListW`.

In summary, at every decision node n that branches on $x = v$: on the left branch, v is removed from `ListW` (if it is present); on the right branch, if v is in `ListW` then $x \neq w$ is posted for every w in `ListW`.

Example 59. Consider the CSP $(X, 1..10, C)$ where $W = \{1, 2, 3, 4\}$ are interchangeable values. Figure 5.6 shows the effect on `ListW` of each particular binding. Importantly, `ListW` only changes whenever the binding involves a value in `ListW`. □

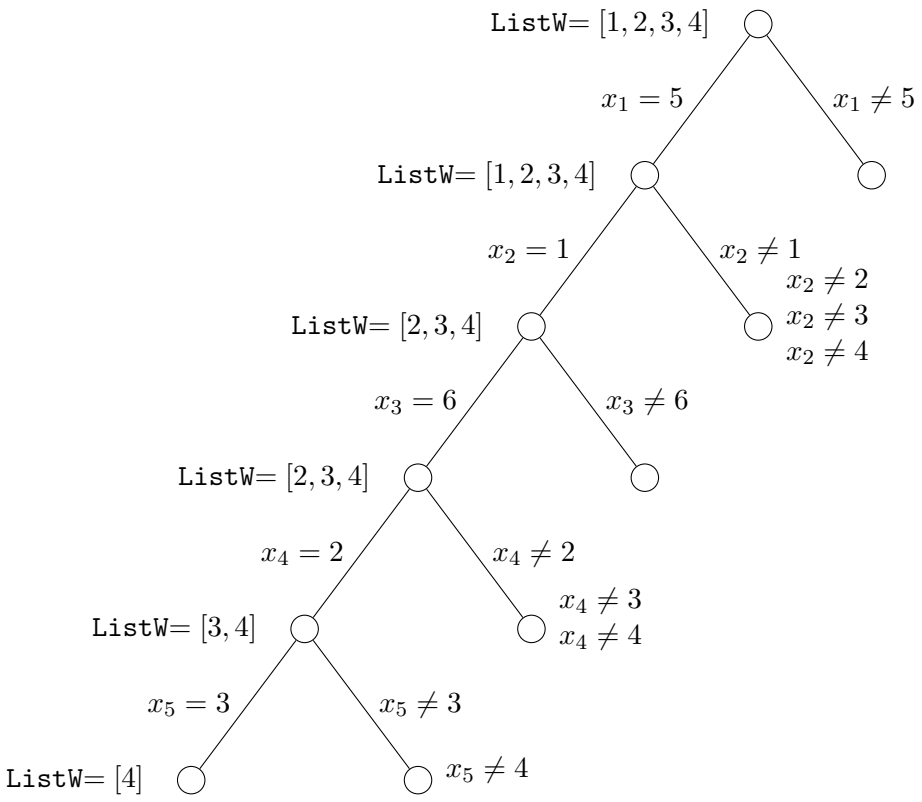


Figure 5.6: Evolution of `ListW` during search for interchangeable values. Symmetry breaking constraints are shown to the right of $x \neq v$ nodes.

LDSB is correct for a set of interchangeable values. As for interchangeable variables, to prove that LDSB breaks interchangeable value symmetries correctly, we rely on the correctness of an $s - excluding$ search tree, shown by Backofen and Will (1999). We show merely that LDSB does a subset of the pruning done by an $s - excluding$ search tree and is therefore correct.

Lemma 9. For the CSP (X, D, C) with the set of interchangeable values $W \subseteq D$, and at a node which branches on $x = a$ and $x \neq a$ where $a \in W$, LDSB prunes on the right branch a subset of the prunings done in an $s - excluding$ search tree.

Proof. First, let $W' = W \setminus L$, where L is the set of values that have been used in a left branch on the path from the root to the current branching node. By definition, a is in either L or W' – it cannot be in neither or both. If $a \in L$ then LDSB does no pruning and is therefore correct. For the remainder of the proof, assume $a \in W'$.

Let G be the group of symmetries due to the interchangeable values W. An $s - excluding$ search tree posts on the right branch $s(C_p) \rightarrow \neg s(x = a)$ for each symmetry $s \in G$, where C_p is the set of left branch constraints posted on the path from the root to current branching node. We assume that every left branch is of the form $Var = Val$ and therefore C_p is a set of literals.

LDSB posts $\neg(x = w)$ for all $b \in W'$. To show that the pruning done by LDSB is a subset of an $s - excluding$ search tree's, we must show that for each w there exists a symmetry $s \in G$ for which $s(C_p)$ holds and $s(v) = w$. To do so, for any given $w \in W'$, we construct a symmetry s such that $s(v) = w$ and show both that $s(C_p)$ holds and that $s \in G$. First, we define $s(v) = w$, $s(w) = v$ and $s(y) = y$ for all $y \in D \setminus \{v, w\}$. It can be seen that neither v nor w is involved in any literal in C_p: if v is involved in a literal of C_p, then $v \in L$ and therefore $v \notin W'$, which contradicts the assumption made earlier; similarly, if w is involved in C_p, then $w \notin W'$ which contradicts the definition of w. Since neither v nor w is involved in C_p, then $s(C_p) = C_p$. Since C_p is known to hold, $s(C_p)$ also holds.

The remaining step is to show that $s \in G$. Since G contains every permutation of the values of W, it must include s (which interchanges only v and w) because v and w both are in W. □

As for interchangeable variables, LDSB optimally efficient for interchangeable values, i.e. it does not post vacuous constraints, does not perform multiple postings, eliminates all broken symmetries, and is complete. Thus, it builds the same search tree as SBDS for a single set of interchangeable values.

Lemma 10. A search tree produced by LDSB for CSP (X, D, C) with interchangeable values W has no distinct symmetric nodes. That is, for any two distinct nodes n_α and n_β, there is no symmetry $\sigma_r \in G(W)$ such that $\sigma_r(\alpha) = \beta$.

Proof. We will prove the above proposition by contradiction. Assume the existence of two distinct nodes in the search tree, n_α and n_β, and some $\sigma_r \in G(W)$ such that $\sigma_r(\alpha) = \beta$. We will show that such an arrangement cannot occur.

Let n_γ be the lowest common ancestor of α and β, with the children of n_γ being $n_{x=v}$ and $n_{x\neq v}$. Without loss of generality, let n_α be a descendant of $n_{x=v}$ (or that node itself) and let n_β be a descendant of $n_{x\neq v}$ (or that node itself). Therefore, $x = v \in \alpha$, $x \neq v \in \beta$ and $x = r(v) \in \beta$.

Clearly, $v \neq r(v)$ because otherwise β would contain both $x = v$ and $x \neq v$. Since $r(v) \neq v$, then $v \in W$. Furthermore, v must be in W' since if there was some $y = v$ before n_γ, then β would contain both $y = v$ and $y = r(v)$, which is impossible.

Let us denote $r(v)$ as e and first consider the case where $e \in W'$. Let $P(W)$ be the set of permutations of W. Since v and e are both in W', and by definition LDSB asserts $x \neq r(v)$ at $n_{x\neq v}$ for all $r \in P(W')$, $x = e$ will be pruned at node $n_{x\neq v}$ and therefore n_β cannot be reached. This means that e cannot be in W'.

Let us now consider the case where $e \notin W'$. This means there is some $a = e$ before n_γ and therefore β contains both $a = e$ and $a = r(e)$. However, $r(e) \neq e$ and therefore this is impossible.

Since e is neither in nor out of W', no such tree exists. □

5.5.3 Interchangeable Variable Sequences

Consider a CSP $P = (X, D, C)$, where S is a set of simultaneously interchangeable variable sequences for P. For any two sequences $s_1, s_2 \in S$, let r be the permutation $(s_1 \leftrightarrow s_2)$, and f_r denote both its induced solution symmetry and (by an abuse of notation) its associated

solution preserving function, previously denoted by σ_{f_r}. Let the search again be at node n with current assignment α, and let $G(S,n)$ be the subgroup of interchangeable sequences of S that are active at node n. Let us also assume that none of the variables in $vars(S)$ appears in α (i.e. $scope(\alpha) \cap vars(S) = \emptyset$). As before, $G(S, root) = G(S,n)$ and therefore all interchangeable sequences in S are active at any node from root to n. In the following we will describe how the program variable `Seqs`, which represents the set S, changes during search.

Suppose now that the first child of n is labelled $x = v$ with associated assignment $\beta = \alpha \cup \{x = v\}$ where $x \in vars(S)$, i.e. x is the first variable in $vars(S)$ to occur in β. Clearly, $f_r(\beta) \neq \beta$ for every r such that $r(x) \neq x$ and, therefore, every such f_r is not active at node $x = d$. At this point, and analogously to what we did for broken interchangeable variable symmetries, one could remove from `Seqs` every list that contains `X`. While doing this is correct, it eliminates too many symmetries, since this time the remaining sequences might not represent symmetries $f_{r'}$ that achieve as much pruning as the reactivated f_r. Instead, we will simply keep all sequences, even if variables in them are ground. This allows our implementation to take advantage of reactivated symmetries.

Example 60. Consider a search where `Seqs=[[X,Y,Z],[A,B,C],[U,V,W]]`, representing (among others) symmetry $r = \langle x, y, z\rangle \leftrightarrow \langle a, b, c\rangle$. At node `X=1` list `Seqs` becomes `[[1,Y,Z],[A,B,C],[U,V,W]]`, at which point r is broken. However, if the next node binds `A` to `1`, r is reactivated. If the sequence `[X,Y,Z]` had been removed from `Seqs`, the symmetry would not be detected. As we will see later, by leaving the sequences intact the reactivated symmetry can be detected from the new instantiation of `Seqs=[[1,Y,Z], [1,B,C], [U,V,W]]`. □

Upon backtracking, the second child of node n will be labelled $x \neq v$. At this point, for any $r \in G(S,n)$ such that $f_r(\beta) \neq \beta$, we add $\neg r(x) = v$ and, thus, prune every part of the subtree symmetric to β under f_r. This is achieved by finding every sequence `S` $\in$ `Seqs` that contains `X`, and comparing `S` to every other sequence `R` $\in$ `Seqs` to determine whether `S` $\leftrightarrow$ `R` is an active symmetry for α. This is true if for every pair of corresponding elements `A` and `B` in `S` and `R`, either variable `A` has the same value as variable `B` or neither variable has a fixed value. For every `R` for which this is true, we add `Y \= v`, where `Y` occurs in `R` in the same position as `X` in `S`.

In summary, at every decision node n that branches on $x = v$: on the left branch, every occurrence of x in `Seqs` is replaced by v; on the right branch, if x is in a sequence S in `Seqs` then every other sequence T is checked to see if the symmetry between S and T is active and if so, $y \neq v$ is posted for the variable y in the position in the sequence corresponding to x.

LDSB is correct for interchangeable variable sequences. As previously, to prove that LDSB breaks interchangeable variable sequence symmetries correctly, we rely on the correctness of an $s - excluding$ search tree, shown by Backofen and Will (1999). We show merely that LDSB does a subset of the pruning done by an $s - excluding$ search tree and is therefore correct.

Consider the CSP $P = (X, D, C)$ with the set S of interchangeable sequences of variables:

$\{\alpha_1, \alpha_2, \ldots, \alpha_n\}$
$\{\beta_1, \beta_2, \ldots, \beta_n\}$
$\{\gamma_1, \gamma_2, \ldots, \gamma_n\}$
$\vdots$

Let G be the group of symmetries due to the interchangeable sequences in S. By an abuse of notation we say that a variable is in S if it is in any sequence in S. At a given node the search branches on a variable x. If $x \notin S$, then LDSB does no pruning and is therefore correct, so we consider only nodes that branch on some $x \in S$.

Lemma 11. For the CSP (X, D, C) with the set S of interchangeable sequences of variables, and at a node which branches on $x = v$ and $x \neq v$ where $x \in S$, LDSB prunes on the right branch a subset of the prunings done in an $s - excluding$ search tree.

Proof. Without loss of generality, let us assume that x is α_i and consider only the symmetry that interchanges the α sequence with the β sequence. LDSB will assert $\beta_i \neq v$ on the right branch only if, for all $1 \leq j \leq n$, $\alpha_j \approx \beta_j$, where $(v \approx w) \equiv (nonground(v) \wedge nonground(w)) \vee (v = w)$. If this property is not true then LDSB does no pruning and is therefore correct; we assume from this point that the property is true.

To show that the pruning done by LDSB is a subset of an $s - excluding$ search tree's, we must show that if the above property is true, then there exists a symmetry $s \in G$ for which $s(C_p)$ holds and $s(\alpha_i) = \beta_i$. To do so, we construct a symmetry s such that $s(\alpha_i) = \beta_i$ and show both that $s(C_p)$ holds and that $s \in G$. First, we define $s(\alpha_i) = \beta_i$, $s(\beta_i) = \alpha_i$ and $s(y) = y$ for all $y \in X \setminus \{\alpha_i, \beta_i\}$.

For $s(C_p)$ to hold, it must be the case that, for every $(y = w) \in C_p$, $s(y = w)$ must also hold. For each such $y = w$, there are two cases: either s maps y to itself, or s maps y to another variable. If s maps y to itself, then $s(y = w)$ obviously holds. Assume then that s maps w to another variable. There are two cases: either $s(y)$ is ground or $s(y)$ is not ground. If $s(y)$ is not ground, then the earlier property is violated: y is ground and must be some α_j (or β_j) while its corresponding variable β_j (or α_j) is not ground. Assume then that $s(y)$ is ground. Then $s(y)$ has the same value as y (by earlier property) and $s(y = w)$ holds because it is the same as $y = w = s(y)$.

The remaining step is to show that $s \in G$. Since G contains every permutation due to S, it must include s (which interchanges only the α and β sequences) because the α and β sequences are in S. □

LDSB is, however, not perfectly efficient: it might post vacuous constraints whenever the two sequences contain the same variable in the same position. This is however a rare case and modifying the algorithm to avoid posting vacuous constraints will often result in a loss of performance (based on an experimental evaluation whose results are not shown). LDSB might also perform multiple postings whenever there is more than one symmetry that maps variable `X` to `Y`. It does not eliminate broken symmetries from its representation, since it might need them to detect reactivated symmetries. And finally, as shown in Figure 5.7, it is incomplete due to its adherence to the shortcut method, in contrast to the cases of variable and value interchangeability. As a result, LDSB and SBDS might not build the same search tree (depending on the choice of variable order followed by the search strategy) whenever simultaneous variable interchangeability is considered.

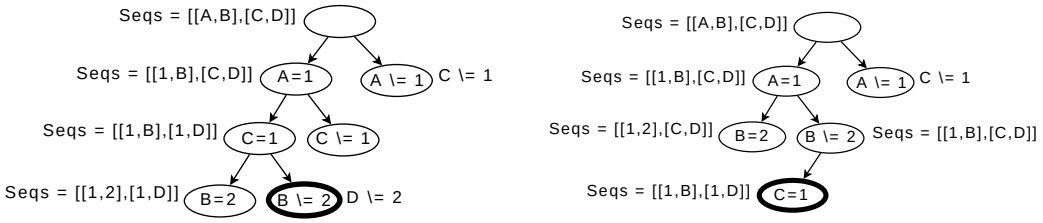


Figure 5.7: Impact of the choice of search heuristic on completeness. The highlighted nodes represent the same constraint store, but LDSB's pruning depends on the search order.

5.5.4 Interchangeable Value Sequences

Consider a CSP $P = (X, D, C)$, where S is a set of simultaneously interchangeable value sequences for P. For any two sequences $s_1, s_2 \in S$, let r be the permutation $(s_1 \leftrightarrow s_2)$, and f_r denote both its induced solution symmetry and (by an abuse of notation) its associated solution preserving function, previously denoted by σ_{f_r}. Let the search again be at node n with current assignment α, and let $G(S, n)$ be the subgroup of interchangeable sequences of S that are active at node n. Let us also assume that none of the values in any sequence of S appears in α. As before, $G(S, root) = G(S, n)$ and therefore all interchangeable sequences in S are active at any node from root to n. In the following we will describe how the program variable `Seqs`, which represents the set S, changes during search.

It is an invariant of `Seqs` that no sequence in `Seqs` contains a value in the range of the current assignment α. That is, every value in `Seqs` has been used in a left branch, and all symmetries between pairs of sequences are active. Therefore, whenever a node n is labelled $x = v$, we remove any sequence containing `v` from `Seqs`. Upon backtracking, to node $x \neq v$, we find every sequence `S` in `Seqs` that contains v, and for every other sequence `R` in `Seqs`, add `X \= w`, where `w` occurs in `R` in the same position as `v` in `S`.

LDSB is correct for interchangeable value sequences. As previously, to prove that LDSB breaks interchangeable value sequence symmetries correctly, we rely on the correctness of an $s - excluding$ search tree, shown by Backofen and Will (1999). We show merely that LDSB does a subset of the pruning done by an $s - excluding$ search tree and is therefore correct.

Consider the CSP (X, D, C) with the set S of interchangeable sequences of values:

$$\{a_1, a_2, \ldots, a_n\}$$
$$\{b_1, b_2, \ldots, b_n\}$$
$$\{c_1, c_2, \ldots, c_n\}$$
$$\vdots$$

We assume the sequences in S are pairwise disjoint. Let G be the group of symmetries due to the interchangeable sequences in S. By an abuse of notation we say that a value is in S if it is in any sequence in S. At a given node the search branches on $x = v$ and $x \neq v$. If $v \notin S$, then LDSB does no pruning and is therefore correct, so we consider only nodes that branch on some $x = v$ where $v \in S$.

Lemma 12. For the CSP (X, D, C) with the set S of interchangeable sequences of values, and at a node which branches on $x = v$ and $x \neq v$ where $v \in S$, LDSB prunes on the right branch a subset of the prunings done in an $s - excluding$ search tree.

Proof. First, let L be the set of values that have been used in a left branch on the path from the root to the current branching node. Let $S' = \{s | s \cap L = \emptyset\}$; in other words, a sequence is in S' only if none of its element values have been used on a left branch. If $v \notin S'$, then LDSB does no pruning and is therefore correct. Assume then that $v \in S'$.

Let G be the group of symmetries due to the interchangeable sequences of values S. An $s - excluding$ search tree posts on the right branch $s(C_p) \rightarrow \neg s(x = v)$ for each symmetry $s \in G$, where C_p is the set of left branch constraints posted on the path from the root to current branching node. We assume that every left branch is of the form $Var = Val$ and therefore C_p is a set of literals.

Assume without loss of generality that value v is a_i. LDSB posts $\neg(x = \rho_i)$ for each sequence $\rho \in S'$. To show that the pruning done by LDSB is a subset of an $s - excluding$ search tree's, we must show that for each ρ there exists a symmetry $s \in G$ for which $s(C_p)$ holds and $s(a_i) = \rho_i$. To do so, for any given $\rho \in S'$, we construct a symmetry s such that $s(a_i) = \rho_i$ and show both that $s(C_p)$ holds and that $s \in G$. First, we define $s(a_i) = \rho_i$, $s(\rho_i) = a_i$ and $s(y) = y$ for all $y \in D \setminus \{a_i, \rho_i\}$. It can be seen that, for any j, neither a_j nor ρ_j is involved in any literal in C_p: if a_j is involved in a literal of C_p, then $a_j \in L$ and therefore $a_j \notin S'$, which contradicts the assumption made earlier; similarly, if ρ_j is involved in C_p, then $\rho_j \notin S'$ which contradicts the definition of ρ_j. Since neither a_j nor ρ_j is involved in C_p, then $s(C_p) = C_p$. Since C_p is known to hold, $s(C_p)$ also holds.

The remaining step is to show that $s \in G$. Since G contains every permutation of the sequences of values in V, it must include s (which interchanges only the a_i sequence and the ρ_i sequence) because both the a_i sequence and the ρ_i sequence are in S. □

LDSB might post vacuous constraints for interchangeable value sequences whenever two sequences contain the same value in the same position. Avoiding this will often result in loss of performance (again based on data not shown). It might also perform multiple postings, whenever there is more than one symmetry that maps value `v` to `w`. It does eliminate broken symmetries from its representation, but it might eliminate too many leading to further incompleteness. This can only happen if the same value appears at the same position in two sequences of `Seqs`. LDSB might thus build a larger tree than SBDS.

5.6 Composing Symmetries

Each LDSB symmetry induces a function that takes as input the current state of the variable domains and a literal, and produces a set of symmetric literals to be pruned. That is, each symmetry s induces a function $s(D', x = v) = L$, where D' is the current variable domains, $x = v$ is the literal and L is the set of symmetric literals. When the search backtracks and follows the right branch of a decision $x = v$, the literals pruned are:

$$\bigcup_{s \in S} s(D', x = v)$$

where S is the set of symmetries. However, these are not the only literals that can be excluded. If f and g are symmetries, then not only can $f(D', x = v)$ and $g(D', x = v)$ be excluded, but also $\{f(D', l) | l \in g(D', x = v)\}$ – that is, any literal resulting from applying first g and then f to $x = v$.

Our implementation finds such literals, for any combination of symmetries, and excludes them. This proves in practice to be vital for certain kinds of problem. The algorithm for doing so is as follows:

```
todo <- [x=v]   \% queue of literals to be processed
done <- [x=v]   \% set of literals already processed
D <- current variable domain
while todo is not empty
  lit <- serve front of todo
  for each symmetry f
    for each literal symlit in f(D, lit)
      if symlit not in done
        add symlit to done
        append symlit to todo
```

In this algorithm, `todo` is a queue of literals to be processed, and `done` is the set of literals to be pruned. Each literal in `todo` is retrieved and its set of symmetric literals computed. If any such symmetric literal has not been seen before, it is added to the set of excluded literals and added to `todo` to be processed. This can be imagined as a breadth-first traversal of a graph, where each literal is a node and there is an edge from literal a to literal b if $b \in s(D', a)$ for some symmetry s.

Example 61. Consider a CSP $(\{x_1, x_2, x_3\}, \{1, 2, 3\}, C)$, that has the following two symmetries: (1) all variables are interchangeable and (2) all values are interchangeable. Let us assume the search first tries $x_1 = 1$ and then backtracks. The above algorithm first processes the literal $x_1 = 1$. The first symmetry adds $x_2 = 1$ and $x_3 = 1$ to both `done` and `todo`; the second symmetry adds $x_1 = 2$ and $x_1 = 3$ to both `done` and `todo`. Next, the literal $x_2 = 1$ is processed. Processing this literal adds to `done` and `todo` the literals $x_2 = 2$ and $x_2 = 3$. Next, the literal $x_3 = 1$ is processed: this adds to `done` and `todo` the literals $x_3 = 2$ and $x_3 = 3$. At this point, `done` contains all of the literals of the problem and consequently every literal is pruned on the right branch $x_1 \neq 1$. See the table below for an illustration.

In the first step, `todo` and `done` are initialised. In the second step, the literal $x_1 = 1$ is served from `todo` and we find the literals that are symmetric to it. These literals $(x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3)$ are added to `done` and to `todo`. These steps are repeated until `todo` is empty.

todo	*done*
$x_1 = 1$	$x_1 = 1$
$x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3$	$x_1 = 1, x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3$
$x_3 = 1, x_1 = 2, x_1 = 3, x_2 = 2, x_2 = 3$	$x_1 = 1, x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3,$ $x_2 = 2, x_2 = 3$
$x_1 = 2, x_1 = 3, x_2 = 2, x_2 = 3, x_3 = 2, x_3 = 3$	$x_1 = 1, x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3,$ $x_2 = 2, x_2 = 3, x_3 = 2, x_3 = 3$
$x_1 = 3, x_2 = 2, x_2 = 3, x_3 = 2, x_3 = 3$	$x_1 = 1, x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3,$ $x_2 = 2, x_2 = 3, x_3 = 2, x_3 = 3$
$x_2 = 2, x_2 = 3, x_3 = 2, x_3 = 3$	$x_1 = 1, x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3,$ $x_2 = 2, x_2 = 3, x_3 = 2, x_3 = 3$
...	...
$\emptyset$	$x_1 = 1, x_2 = 1, x_3 = 1, x_1 = 2, x_1 = 3,$ $x_2 = 2, x_2 = 3, x_3 = 2, x_3 = 3$

□

Note that this algorithm does not truly compose symmetries, but rather applies symmetries repeatedly to literals. This difference is important for efficiency: composing even a small number of symmetries can lead to exponentially large sets of symmetries, but this algorithm only needs to compute at most a quadratic ($|X| \times |D|$) number of literals, while obtaining the same result set of literals.

5.7 Implementation

LDSB has been implemented as a module for the ECL^{i}PS^e (Wallace et al., 1997) constraint programming platform and for the Gecode (Gecode Team, 2006) constraint solving system. The only requirement for a programmer to use LDSB is to specify a set of symmetries of the problem. The symmetry breaking search in LDSB replaces the branching part where $x = v$ and $x \neq v$ constraints are posted. Whenever the search would post an $x = v$ constraint, LDSB updates the current state of the symmetries, and whenever it would backtrack to the $x \neq v$ constraint, LDSB adds constraints to do the symmetric pruning. Note that it does not interfere with the selection of x and v, so any variable or value ordering can be used.

Example 62. The CSP $(\{x_1, x_2, x_3\}, \{1, 2, 3\}, \mathit{all\_different}(\{x_1, x_2, x_3\}))$ has interchangeable variables and interchangeable values. It can be easily represented in ECL^{i}PS^e with LDSB as follows, where LDSB-specific parts are marked with comments.

```
:- lib(ic).
:- lib(ldsb).     % Load LDSB library.
problem :-
  Vars = [](X1,X2,X3),
  Vars #:: 1..3,
  alldifferent(Vars),
```

```
  % Initialise LDSB and give symmetries.
  ldsb_initialise(Vars, [ variable_interchange([X1,X2,X3]),
                          value_interchange([1,2,3]) ]),
  % Search with LDSB.
  search(Vars, 0, input_order, ldsb_indomain, complete, []).
```

To perform the search without symmetry breaking, the call to the search procedure would instead be:

```
search(Vars, 0, input_order, indomain, complete, []).
```

A custom search strategy can be implemented using the `ldsb_indomain` predicate, which instantiates a single variable, and the `ldsb_try` predicates, which attempts to assign a single value to a variable.

Common symmetries, including all variables or all values being interchangeable and row/column symmetries, can be specified with special keywords. For example, the above initialisation can be replaced with:

```
ldsb_initialise(Vars, [variables_interchange, values_interchange])
```

The same problem can be expressed in Gecode, but for brevity we show only the relevant parts instead of the whole program.

```
#include "ldsb.hh"
  [...]
Problem() : vars(*this, 3, 1, 3)
{
    ViewArray<IntView> xv(*this, VarArgArray<IntVar>(vars));
    distinct(*this, vars);

    // Create vector of indices for variable symmetry.
    vector<int> indices(3);
    indices.push_back(0); indices.push_back(1); indices.push_back(2);
    // Create vector of values for value symmetry.
    vector<int> values(3);
    values.push_back(1); values.push_back(2); values.push_back(3);

    // Create vector of symmetries to use with LDSB.
    vector<SymmetryPtr> s;
    s.push_back(SymmetryPtr(new VariableSymmetry(indices)));
    s.push_back(SymmetryPtr(new ValueSymmetry(values)));

    // Pass symmetries to LDSB branching algorithm.
    ldsb_branch(*this, xs, INT_VAR_NONE, INT_VAL_MIN,
                VarBranchOptions::def, ValBranchOptions::def, s);
}
```

Note that the `ldsb_branch` function accepts the same options as Gecode's own branch method, allowing any variable and value ordering to be used with LDSB. □

The kernel of the search ECL^{i}PS^e implementation is as follows – the Gecode implementation follows the same principles:

```
try(Var, Val) :-
     (
        Var = Val,
        update_symmetries(Var, Val)
     ;
        compute_prunings(Var, Val, Prunings),
        Var \= Val,
        prune(Prunings)
     ).
```

This predicate first attempts to assign the value *Val* to *Var*. This corresponds to the left branch of the search tree, wherein the symmetries are updated where applicable. Depending on the symmetries of the problem, which are accessible via global state, this may result in *Var* or *Val* being removed from interchangeable variable or value sets.

If this branch is unsuccessful – either immediately, or after some further search in the subtree of the left branch – the right branch is tried. This involves computing all of the literals that are symmetric to $Var = Val$ (i.e. the function $s(D', (Var = Val))$ for every $s \in S$), and then excluding those literals along with the standard right branch constraint $Var \neq Val$.

The `update_symmetries` predicate updates the interchangeable variable, interchangeable value and interchangeable value sequence lists by simply removing `Var`, `Val` or the sequence containing `Val` respectively from them, if present. Traversal of these lists is fast in practice. Note that interchangeable variable sequences symmetries are not modified by it, except for the fact that variables in the sequences might become ground.

The predicate `compute_prunings` operates on the symmetries as they appeared before the decision point, as follows. It looks for the first interchangeable variable list `W` that contains `Var`. If it finds one, it adds `Y` $\neq$ `Val` to `Prunings` for every other `Y` in `W`. It also looks for the first interchangeable value list `M` that contains `Val`. If it finds one, it adds `X` $\neq$ `d` for every other `d` in `M`. For interchangeable variable sequences, it accesses a lookup table that indicates which interchangeable variable sequence symmetries the `Var` is in and, for each such symmetry `Seqs`, in which sequence `S` of `Seqs` the `Var` is in, and its position. It then compares each such `S` with every other sequence in `Seqs`. Having direct access from `Var` to the list of symmetries is crucial for performance. This is achieved by creating the lookup table for each variable before the search begins. For every new pruning added to `Prunings`, a recursive call to `compute_prunings` is performed to determine possible new prunings achieved by composition, until no new prunings are obtained. Finally, predicate `prune` takes a list of elements of the form `Y` $\neq$ `Val` and posts them.

In Gecode we add the update and pruning steps to the system's own depth-first search engine, in similar locations to the ECL^{i}PS^e implementation.

There are several promising variations to the current implementation. One is to represent interchangeable variable (or value) sequences pairwise – i.e. represent each pair of sequences in `Seqs` separately. While this results in a quadratic number of extra symmetries, it allows permanently inactive pairs to be deleted from the list. Additionally, when two variables in the same position of their respective sequences are bound to the same value, the variables can be removed from the sequences. This also eliminates the potential incompleteness of simultaneous value interchangeability. Another variation is not to remove variables from the interchangeable variable lists as they become ground. This would make updates faster but lookups slower (since the lists will be longer).

5.8 Experimental Results for ECL^{i}PS^e

In this section the empirical performance of LDSB is demonstrated by comparing the running time of LDSB with other dynamic symmetry breaking methods. In particular, the experiments compare LDSB with four other methods: no symmetry breaking, a generic and incomplete form of SBDS and two complete methods based on computational group theory, GAP-SBDS and GAP-SBDD. Each method is tested using both a naive variable order and the first-fail heuristic. These experiments were run using the IC constraint solver of ECL^{i}PS^e version 6.0 #80 and the GAP system version 4.4.12.

The incomplete form of the SBDS method is provided with the same symmetries as LDSB. To do so generically it is necessary to convert LDSB's set of symmetry patterns into symmetry functions in SBDS. However, it is impractical to have one function for each symmetry in the group, since this would result far too many functions. Instead, each set of interchangeable items (whether variables, values or sequences of same) is converted into a set of symmetry functions, one per transposition of items in the set. Also, to keep to a practical number of symmetry functions, symmetries are not combined.

The experiments were run on a cluster of quad-core Intel Xeon E5310 1.66GHz CPUs with a limit of 1GB of memory, using Linux kernel version 2.6.18. Although different benchmarks may have been run on different machines in the cluster, for the purpose of comparing the different methods, all methods for a given single benchmark were run on the same CPU.

The benchmarks are divided into two sets. The first set of benchmarks includes constraint satisfaction problems often used for symmetry detection/breaking. We will call this set the "satisfaction" benchmarks. It includes the following problems, whose details can be seen in Appendix A:

- **bibd**$((v, b, k, r, \lambda)$: balanced incomplete block design of v objects into b blocks of exactly k objects, where each object is in r blocks, and every two objects occur together in λ blocks. The objects are interchangeable and the blocks too, giving $v!b!$ symmetries, all represented in LDSB.

- **golf**(w, g, p): social golfers to arrange gp golfers each week into g groups of p players over w weeks, with no two players appearing in the same group twice. The weeks are interchangeable, the players are too and, within each week, the groups are interchangeable giving $w!(gp)!(g!)^w$ symmetries, all in LDSB.

- **graceful**(m, n): graceful graph where the edges (a, b) of the $K_m \times P_n$ graph are labelled $|a - b|$, and no two edges have the same label. The corresponding vertices in each clique are simultaneously interchangeable, the order of the cliques is reversible (represented via simultaneously interchangeable values), and the values are too, giving $4m!$ symmetries, all in LDSB.

- **latin**(n): latin square with $2(n!)^3$ of the $6(n!)^3$ symmetries represented in LDSB.

- **magicsquare**(n): magic $n \times n$ square filled with 1 to n^2 numbers such that all numbers are used and all rows, columns and two diagonals have the same sum. We use the symmetries of the square and the reversible value symmetries, giving 16 symmetries, all in LDSB.

- **nn_queens** (n): an $n \times n$ chessboard coloured with n colours, so that a pair of queens in any two squares of the same colour do not attack each other. The symmetries are those of the chessboard, plus the colours are interchangeable, giving $8n!$ symmetries, all in LDSB.

- **queens**(n): n queens placed on a chessboard with no two queens attacking each other. Modelled with one variable per queen whose value is the queen's row. The symmetries are those of the chessboard (i.e., 8 symmetries) with only 4 in LDSB.

- **queens_bool**(n): same as above but with a Boolean matrix model. All 8 symmetries are in LDSB.

- **steiner**(n): steiner triples to find $\frac{n(n-1)}{6}$ triples of distinct integers from 1 to n, with any two triples having at most one element in common. The triples are interchangeable and the values too, giving $n!^{\frac{(n(n-1)}{6}}!$ symmetries, all in LDSB.

The second set of benchmarks are instances of the following optimisation problems, with all their symmetries represented in LDSB. We will call these the "optimisation" benchmarks. Further details are provided in Appendix A.

- **Graph Colouring**: where the task is to find the chromatic number of a graph. The instances are randomly generated following (Law et al., 2007) to have partitions of symmetric variables and values: n nodes are partitioned (with $k = 8$ set to the maximum size) so that each partition is either an independent or a complete graph, and each subgraph with nodes in two partitions is either an independent or a complete bipartite graph. Instances are divided into classes according to (1) how the partition sizes are distributed ("uniform" or "biased"), (2) the number n of nodes, and (3) the proportion q of partitions that are complete subgraphs. There are 20 instances of each class. Colours are interchangeable (giving $n!$) and nodes too within each partition.

- **Concert Hall Scheduling**: choose among n applications specifying a period and an offered price to use k identical concert halls, to maximise profit. The instances are randomly generated following (Law et al., 2007): k is set to 8 and applications are generated in partitions of uniformly distributed size, with all applications in a

partition being identical. There are 20 instances for each size. Concert halls are interchangeable (k! symmetries) and applications within each partition are too.

- **Steel Mill Slab Design**: arrange n orders of slabs of a particular size and colour to minimise wasted space when cutting a given set of slabs of predetermined size. The sum of the sizes of orders on each slab must not exceed the slab's capacity, and the number of distinct colours of orders on one slab must be at most two. The instances are taken from CSPLib(Gent and Walsh, 1999). The slabs are interchangeable (n! symmetries).

5.8.1 Discussion

The experimental results in ECL^{i}PS^e for the satisfaction benchmarks are shown in Tables 5.1, 5.2, 5.3 and 5.4. For each benchmark and method, the tables show the time taken in seconds and the number of nodes of the decision tree explored to find the first/all solutions using the first-fail and input-order variable ordering heuristics. A blank entry indicates that the search timed out at the 600 second limit or memory was exhausted. The table headings are "N" for no symmetry breaking, "GS" for GAP-SBDS, "GD" for GAP-SBDD, "IS" for (possibly) incomplete SBDS, and "L" for LDSB. For each category, the best result is shown in bold. For some combinations of search strategy and problem, no method was able to finish within the time limit. These have been included in the tables for comparison against other search strategies.

The results for satisfaction benchmarks show that although in some cases no symmetry breaking is the best method (to find the first solution), LDSB is never more than 25% slower and in many cases is faster, while no symmetry breaking can perform very badly. When compared to the complete symmetry breaking methods, LDSB is uniformly faster than GAP-SBDS (and often orders of magnitude faster) due to its lower overhead. The same happens for GAP-SBDD with the exception of two benchmarks. LDSB also has a low memory overhead compared to the GAP-based methods which have to deal with large data structures when the symmetry group is large. LDSB's performs worst when eliminated symmetries become reactivated (bibd and all solutions for queens_bool). In this case the incomplete SBDS method performs best, with LDSB taking up to twice as long. It is clear from the results that LDSB is the most consistent method (often returns best or near best times).

Let us now discuss the optimisation benchmarks. Figure 5.8 shows the results for the 440 concert hall optimisation instances and for 440 out of a total of 2520 graph colouring instances. The horizontal axis shows the instance size (nodes or offers) while the vertical shows the ratio between the sum of LDSB's running time and the sum of each method's running time over all instances solved by both LDSB and that method. For example, 0.25 indicates that LDSB's time was one quarter of the other method's for the instances solved by both. These problems effectively cannot be solved without symmetry breaking and so we do not show the results for no symmetry breaking. We chose to show only two plots for the colouring problem, since the results from the others are quite similar (the ratio is never higher that 0.4 with incomplete SBDS being the only method to gets close to 0.4, the others being often bellow 0.1). Overall, LDSB solved 2099 of the graph colouring instances as compared to 140 for GAP-SBDS, 1152 for GAP-SBDD and 1884 for incomplete SBDS.

	Time (seconds)					Nodes				
Problem	N	GS	GD	IS	L	N	GS	GD	I	L
bibd [21, 21, 5, 13, 3]	-	-	-	**70.5**	123.5	-	-	-	**5627**	27866
bibd [25, 20, 9, 7, 3]	-	-	-	11.0	**3.9**	-	-	-	**709**	1448
golf [5, 5, 4]	-	-	-	88.5	**47.5**	-	-	-	**6332**	6396
golf [5, 6, 4]	342.9	-	-	63.3	**34.7**	27048	-	-	**5121**	**5121**
graceful [3, 4]	373.5	228.1	276.3	154.8	**153.8**	202454	**82206**	**82206**	**82206**	**82206**
graceful [5, 2]	-	434.2	452.7	316.3	**304.0**	-	**52966**	**52966**	57532	57532
latin 40	**19.4**	-	-	38.1	21.1	**1369**	-	-	**1369**	**1369**
latin 60	97.0	-	-	-	**95.6**	**3278**	-	-	-	**3278**
magicsquare 5	**0.1**	1.3	1.3	**0.1**	**0.1**	**763**	**763**	**763**	**763**	**763**
magicsquare 6	**7.9**	188.8	232.1	11.3	9.0	**71652**	**71652**	**71652**	**71652**	**71652**
nn_queens 7	**0.0**	0.5	0.4	0.1	**0.0**	**16**	**16**	**16**	**16**	**16**
nn_queens 8	-	153.1	177.7	**17.2**	17.8	-	**76073**	76079	149579	149579
nn_queens 9	-	-	-	-	-	-	-	-	-	-
queens 111	**1.4**	122.1	76.2	1.5	1.5	**4570**	**4570**	**4570**	**4570**	**4570**
queens 113	0.5	41.3	31.1	**0.4**	0.5	**1378**	**1378**	**1378**	**1378**	**1378**
queens_bool 26	**4.0**	85.9	220.5	14.3	**4.0**	**11645**	**11645**	**11645**	**11645**	**11645**
queens_bool 28	**28.7**	-	-	109.1	31.5	**90095**	-	-	**90095**	**90095**
steiner 9	1.4	7.8	4.6	0.5	**0.2**	660	**22**	**22**	97	140
steiner 15	0.3	-	-	1.1	**0.2**	**35**	-	-	**35**	**35**

Table 5.1: Benchmark Results, first-fail variable ordering, first solution. 600 second time-out.

	Time (seconds)					Nodes				
Problem	N	GS	GD	IS	L	N	GS	GD	I	L
bibd[13,13,7,7,2]	-	5.6	4.2	0.3	**0.2**	-	**28**	34	33	34
bibd[21,21,5,13,3]	-	-	-	**66.2**	119.2	-	-	-	**5627**	27866
golf [5, 5, 4]	-	-	-	-	-	-	-	-	-	-
golf [5, 6, 4]	-	-	-	-	-	-	-	-	-	-
graceful [3, 4]	-	-	-	-	-	-	-	-	-	-
graceful [4, 2]	540.0	8.2	9.1	23.2	**6.2**	399043	**2262**	2285	11710	3100
graceful [5, 2]	-	492.3	544.7	-	**402.1**	-	**64132**	64168	-	97502
latin 6	-	123.5	**8.7**	10.7	11.2	-	**110**	117	14067	17102
latin 7	-	-	-	-	-	-	-	-	-	-
magicsquare 4	25.1	20.3	37.0	3.4	**1.9**	279180	27235	31052	30326	**18850**
magicsquare 5	-	-	-	-	-	-	-	-	-	-
nn_queens 7	384.4	1.1	1.1	**0.1**	**0.1**	4324319	**432**	437	863	863
nn_queens 8	-	146.7	184.0	17.4	**17.2**	-	**76073**	76079	149579	149579
queens 14	166.1	338.7	587.8	96.0	**64.4**	3195965	**552180**	552195	1408475	992027
queens 15	-	-	-	598.2	**380.0**	-	-	-	8505109	**5654808**
queens_bool 12	16.8	38.2	85.5	**9.6**	15.8	125275	**29415**	29419	50149	105370
queens_bool 13	87.7	208.2	542.8	**49.7**	83.5	635073	**145460**	145479	253892	539244
steiner 9	-	26.9	**16.1**	29.5	19.0	-	**85**	91	7528	13831
steiner 15	-	-	-	-	-	-	-	-	-	-

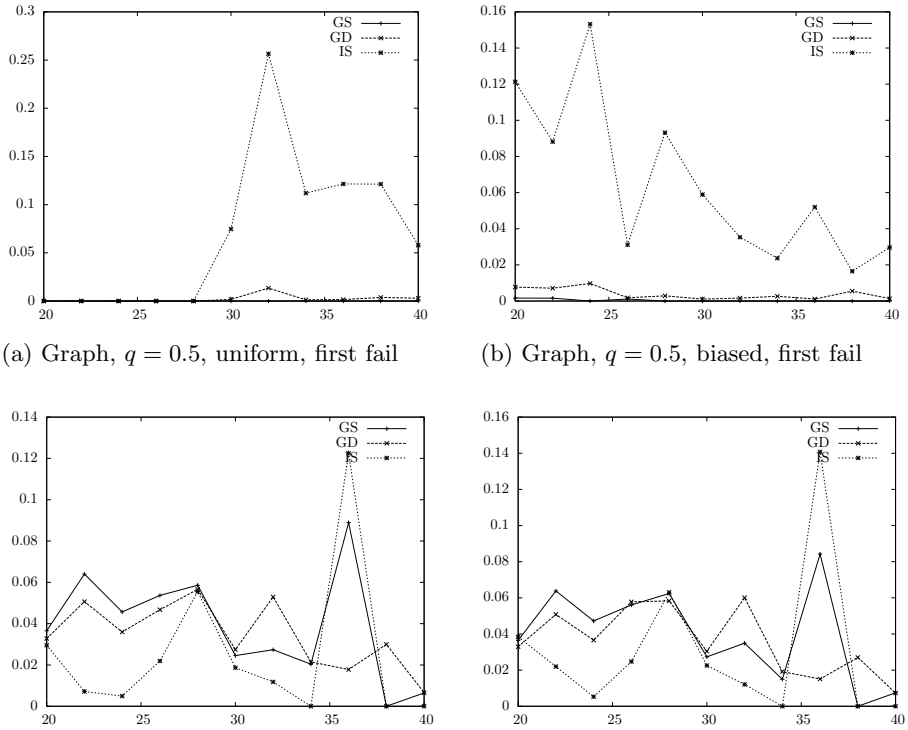
Table 5.2: Benchmark Results, first-fail variable ordering, all solutions. 600 second time-out.

	Time (seconds)					Nodes				
Problem	N	GS	GD	IS	L	N	GS	GD	I	L
bibd[21,21,5,13,3]	-	-	-	**62.3**	112.1	-	-	-	**5627**	27866
bibd[25,20,9,7,3]	-	-	-	10.6	**3.5**	-	-	-	**709**	1448
golf [5, 5, 4]	-	-	-	84.0	**46.2**	-	-	-	**6332**	6396
golf [5, 6, 4]	335.2	-	-	62.7	**34.1**	27048	-	-	**5121**	**5121**
graceful [3, 4]	11.5	17.9	22.3	11.2	**11.0**	8199	**4799**	**4799**	**4799**	**4799**
graceful [5, 2]	-	-	-	-	-	-	-	-	-	-
latin 40	18.4	-	77.9	35.5	**17.9**	**1222**	-	**1222**	**1222**	**1222**
latin 60	**86.3**	-	542.9	-	**86.3**	**2952**	-	**2952**	-	**2952**
magicsquare 5	**0.5**	7.2	10.4	0.6	0.6	**5497**	**5497**	**5497**	**5497**	**5497**
magicsquare 6	-	-	-	-	-	-	-	-	-	-
nn_queens 7	**0.0**	0.5	0.4	**0.0**	**0.0**	**27**	**27**	**27**	**27**	**27**
nn_queens 8	-	228.6	251.4	**17.2**	17.4	-	**104007**	104013	207054	207054
nn_queens 9	-	-	-	-	-	-	-	-	-	-
queens 26	**5.3**	85.0	92.4	5.7	5.8	**55591**	**55591**	**55591**	**55591**	**55591**
queens 28	**40.5**	644.2	-	46.2	46.4	**417043**	**417043**	-	**417043**	**417043**
queens 111	-	-	-	-	-	-	-	-	-	-
queens 113	-	-	-	-	-	-	-	-	-	-
queens_bool 26	**2.2**	85.4	211.2	12.7	2.7	**11645**	**11645**	**11645**	**11645**	**11645**
queens_bool 28	**17.7**	-	-	96.5	20.4	**90095**	-	-	**90095**	**90095**
steiner 9	1.4	7.7	4.8	0.4	**0.2**	660	**22**	**22**	97	140
steiner 15	0.3	-	-	1.2	**0.2**	**35**	-	-	**35**	**35**

Table 5.3: Benchmark Results, input-order variable ordering, first solution. 600 second timeout.

	Time (seconds)					Nodes				
Problem	N	GS	GD	IS	L	N	GS	GD	I	L
bibd[13,13,7,7,2]	-	6.2	4.6	0.4	**0.2**	-	**28**	34	33	34
bibd[21,21,5,13,3]	-	-	-	**65.0**	113.1	-	-	-	**5627**	27866
golf [5, 5, 4]	-	-	-	-	-	-	-	-	-	-
golf [5, 6, 4]	-	-	-	-	-	-	-	-	-	-
graceful [3, 4]	-	-	-	-	-	-	-	-	-	-
graceful [4, 2]	514.8	17.4	20.5	45.2	**13.0**	367259	**5223**	5246	19633	5939
graceful [5, 2]	-	-	-	-	-	-	-	-	-	-
latin 6	-	133.1	16.5	**10.6**	11.2	-	**122**	129	13955	17139
latin 7	-	-	-	-	-	-	-	-	-	-
magicsquare 4	33.5	39.9	76.7	8.2	**3.7**	412445	57358	57411	72342	**35055**
magicsquare 5	-	-	-	-	-	-	-	-	-	-
nn_queens 7	340.5	1.3	1.3	**0.1**	**0.1**	5412959	**602**	607	1079	1079
nn_queens 8	-	218.7	271.8	17.3	**17.1**	-	**104007**	104013	207054	207054
queens 12	5.9	18.0	25.5	4.1	**3.0**	146101	**31070**	31079	68800	55142
queens 13	33.4	91.7	167.2	22.5	**16.1**	756885	**159139**	159151	353528	279679
queens 14	190.8	559.7	-	128.1	**91.3**	4198219	**875178**	-	1961859	1554840
queens 15	-	-	-	-	**545.5**	-	-	-	-	**9066545**
queens_bool 11	2.3	7.7	16.2	**1.6**	2.2	26194	**6274**	**6274**	10371	21299
queens_bool 12	11.2	36.8	86.6	**7.4**	11.2	125275	**29415**	29419	50149	105370
queens_bool 13	59.4	197.0	536.2	**37.6**	58.6	635073	**145460**	145479	253892	539244
steiner 9	-	26.9	**16.0**	30.8	19.1	-	**85**	91	7528	13831
steiner 15	-	-	-	-	-	-	-	-	-	-

Table 5.4: Benchmark Results, input-order variable ordering, all solutions. 600 second timeout.



(a) Graph, $q = 0.5$, uniform, first fail

(b) Graph, $q = 0.5$, biased, first fail

(c) Concert hall, input order

(d) Concert hall, first fail

Figure 5.8: Graph Colouring (a and b) and Concert Hall Scheduling (c and d)

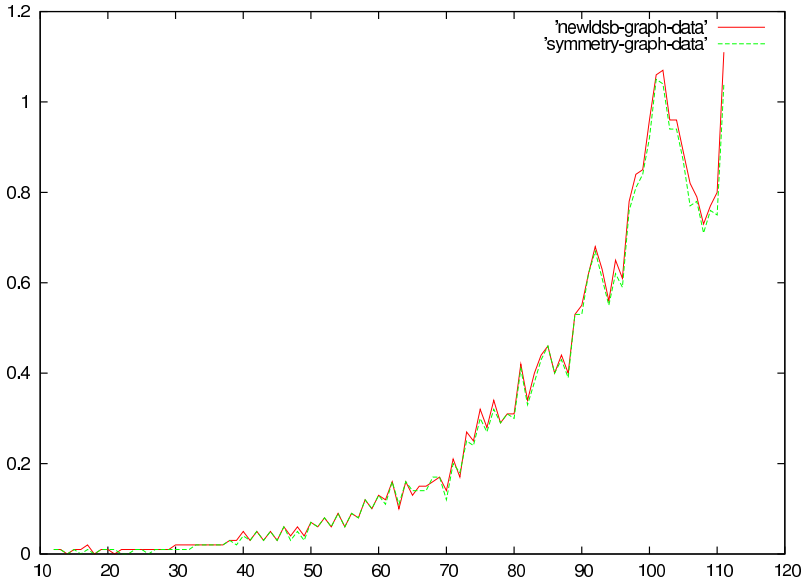


Figure 5.9: Steel Mill Slab Design

Of the concert hall scheduling instances, LDSB solved 340 as compared to 227 for GAP-SBDS, 234 for GAP-SBDD and 147 for incomplete SBDS. The time comparison plots (shown and not shown) indicate that, for the instances where a comparison can be made, LDSB is always faster on average and usually by a factor of at least 3.

5.9 Experimental Result for Gecode

In addition to the experiments comparing LDSB in ECL^{i}PS^e with other dynamic methods, we have performed a second experiment that provides a proof of concept of LDSB's Gecode implementation by comparing LDSB's execution against a hand-crafted search strategy for many instances of a single benchmark.

For the proof of concept of LDSB's Gecode implementation we chose the steel mill slab design problem (problem 38 in CSPLib (Gent and Walsh, 1999)), for which Gargani and Refalo (2007) noted that static symmetry breaking caused a loss of performance for their model. Following this Van Hentenryck and Michel (2008) showed that a customised search strategy with the same model could break most of the symmetry and yield good performance. We have used Gargani and Refalo's model with LDSB and found that it performs remarkably similarly to Van Hentenryck and Michel's customised search.

This result, while preliminary, is encouraging as it shows that our generically applicable method can perform as well as a problem-specific solution. Our method only requires the user to provide the symmetries, rather than to hand-craft a custom symmetry breaking search algorithm themselves.

5.10 Conclusion

In this chapter we have presented a method of symmetry breaking, Lightweight Dynamic Symmetry Breaking, whose aim is to be used as a default symmetry breaking method. To achieve this it must be able to handle commonly occurring symmetries efficiently and must, in almost all cases, reduce the time taken to solve problems. It must also be automatic, so that the user need only provide the symmetries of the problem and not be burdened with the task of providing symmetry breaking constraints or dominance checking functions. It should also be generic: it should not interfere with whichever search heuristics the programmer wishes to use. For efficiency and predictability it must use only simple algorithms and avoid the potential expense of computational group theory.

The results presented suggest that LDSB has achieved these goals. Its consistency in all of the benchmarks indicates that a programmer can use LDSB by default, without worrying that its running time might be inordinately high as can be the case with some complete methods. Also, it can be used simply by specifying the symmetries of the problem. The overhead induced by the symmetry breaking is almost always repaid handsomely by the savings gained, and even when that is not the case, the overhead is low. However, even LDSB's performance is not as good as static symmetry breaking when conditions suit the latter (i.e. a small set of effective constraints can be found that do not conflict with the search order).

Although we have done some optimisations to the representation and its implementation, there is still much room for improvement. The method can be extended to handle variable-value symmetries, either by supporting them directly or perhaps by transforming them into variable symmetries. It would also be interesting to see if the focus on common symmetry patterns can be applied to SBDD as we have done here for shortcut SBDS, and whether better results may come by considering only a subset of a problem's symmetries.

Chapter 6

Generality of Dynamic Symmetry Breaking

6.1 Introduction

In the previous chapter we have presented a new dynamic symmetry breaking method called LDSB. A primary motivation when designing our method is for it to be used in as many situations as possible. However, as presented, the LDSB method imposes some restrictions both on the search and on the constraint solver. Regarding search, while LDSB does not impose restrictions on the user as to the variable or value ordering, it does require the search to make binary decisions of the form ($var = val$) and ($var \neq val$). Therefore, it is not possible for the search to use other common forms of unary constraints, such as $var \leq val$, as branching constraints.

Regarding the constraint solver, the correctness of LDSB relies on the properties of the propagation performed by the constraint solver. This is because at certain points during the search LDSB examines the current domains of the variables involved in symmetries, domains that are maintained and modified by the constraint solver. Since solvers vary in the strength of their propagation, equivalent search decisions may give rise to different domains on different solvers.

In this chapter, we present a preliminary exploration of an extension of LDSB to use domain splitting instead of solely equality branching constraints. We examine the effect of this on how symmetry information is maintained during the search and on the efficiency of detecting when symmetric subtrees should be excluded. In addition, we consider the interplay between LDSB and its constraint solver, and impose some simple minimum requirements on a solver for correct use in conjunction with dynamic symmetry breaking.

We have already demonstrated that LDSB behaves robustly for a search using instantiation branching constraints. Here we show that it is robust with domain splitting as well. Our results also show that domain splitting is indeed a worthwhile feature to support, as it performs consistently better on our sample problem. Parts of this chapter have previously been published in (Mears et al., 2008c).

6.2 Background

Recall the $\mathcal{S}$-*excluding* search tree (Backofen and Will, 1999), described in the previous chapter (see Section 5.2.3). The diagram of a search node in an $\mathcal{S}$-*excluding* search tree is shown again in Figure 6.1.

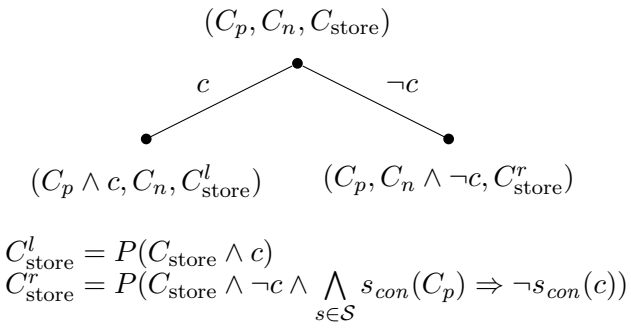


Figure 6.1: A node in an $\mathcal{S}$-*excluding* search tree Backofen and Will (1999).

It is easy to see that the $\mathcal{S}$-*excluding* search tree can easily be instantiated for a search that uses domain-splitting branching constraints. One needs only to instantiate c to be of the form $x \leq v$, where x is a variable and v is a value. The difficulty lies in the definition of the function s_{con}, which must be defined to act on the branching constraints. With instantiation branching, each branching constraint is simply a literal and therefore a symmetry can act as s_{con} directly, since symmetries are defined as permutations of literals. For domain-splitting, the branching constraint is not simply a literal, and s_{con} requires a different definition.

6.3 SBDS with Domain Splitting

Recall that the SBDS algorithm (on which LDSB is based), described in Section 5.2.3, is an instance of an $\mathcal{S}$-*excluding* search tree. SBDS adds to the right branch of each node constraints of the form:

$$A \;\&\; s(A) \;\&\; var \neq val \Rightarrow s(var \neq val) \tag{6.1}$$

where s is a symmetry and A is the current partial assignment.

Gent and Smith (2000) state that SBDS could be extended to other forms of branching constraint. In particular, if the search was to use domain splitting (that is, a decision between $var \leq val$ and $var > val$), then the general SBDS constraint would become:

$$A \;\&\; s(A) \;\&\; var > val \Rightarrow s(var > val) \tag{6.2}$$

where s and A are as before. This constraint is equivalent to that of the $\mathcal{S}$-*excluding* search tree, except that the antecedent is the symmetry of the partial assignment instead of the conjunction of arbitrary branching constraints.

Surprisingly, the above extension has two problems. First, symmetry s must be applied to the constraint $var > val$, a task that may not be straightforward. Second, and more important, the extension is in general incorrect.

Example 63. Consider a CSP $(\{x, y\}, \{1..6\}, \emptyset)$ where there is a variable symmetry $x \leftrightarrow y$. Assume that the search uses domain splitting and first branches on x (see Figure 6.2) with the left branch of the root being labelled $x \leq 3$ and the right branch $x > 3$. Consider the effect of the extended SBDS constraint for the symmetry $x \leftrightarrow y$ on the right branch: $A = \emptyset$ and therefore the antecedent is true and $s(x > 3) \equiv y > 3$ is posted.

Now consider the right branch of the node marked α in Figure 6.2. The partial assignment A is still empty, so A and $s(A)$ hold trivially. The constraint $s(y > 3) \equiv x > 3$ is posted, which in conjunction with the branching constraint $x \leq 3$ leading to α causes the search to fail. This is incorrect as some solutions are not found: for example, the solution $\{(x = 1), (y = 5)\}$ is not found (and neither is the solution symmetric to it).

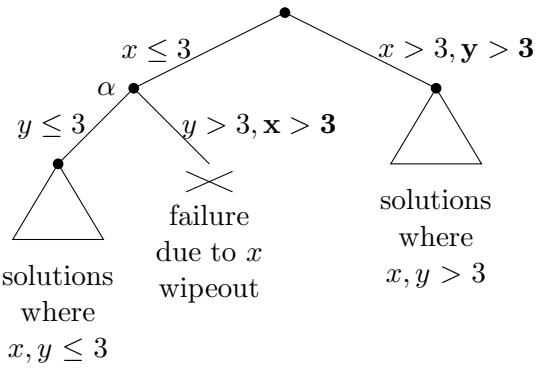


Figure 6.2: SBDS with domain splitting. Search decisions are on each edge; additional symmetry breaking constraints are in bold.

□

The erroneous symmetry breaking is due to SBDS's reliance on "the current partial assignment," i.e. with the fact that at node α, SBDS still considers x and y to be symmetric despite the search having made them asymmetric (by branching on x at the root). In other words, SBDS (and LDSB) does not always work if the search can break symmetries without making variables ground.

The $\mathcal{S}$-*excluding* search tree defined by Backofen and Will does not suffer from this problem because it operates on the branching constraints rather than on the partial assignment.

6.3.1 Symmetries of Branching Constraints

Let us now return to the problem of extending symmetries to act on constraints. The question regards the meaning of expressions such as $s(var > val)$ – if s is defined as a mapping of literals, how does it act on $var > val$? For a variable symmetry such as the one in the previous example the meaning is clear, but in other cases it is not straightforward.

To find a general answer, we must return to the underlying definition of an $\mathcal{S}$-*excluding* search tree.

For a given symmetry s which acts on literals, we must find a corresponding symmetry s_{con} which acts on constraints. Let us define $Sol(c)$ as the set of allowed assignments of a constraint c. The definition of s_{con} must be such that $Sol(s_{con}(c)) = s(Sol(c))$ (following the definition by Backofen and Will (1999)). By an abuse of notation we will also allow s_{con} to act on conjunctions of constraints, such that $s_{con}(c_1 \wedge c_2) = s_{con}(c_1) \wedge s_{con}(c_2)$. In addition, we would like to find an s_{con} such that $s_{con}(c)$ can be represented by a primitive constraint, such as $x = v$ or $x < v$. Primitive constraints are useful because they can be matched syntactically with one another efficiently. In order to determine whether a symmetry s is active at a given point in the search, it is necessary to tell whether $s_{con}(C)$ is entailed, for some conjunction C of constraints. Using primitive constraints allows us to detect entailment of the conjunction of primitive constraints efficiently.

Rather than attempting to define s_{con} for arbitrary symmetries and constraints, we will consider particular symmetries that are known to be simple and common. Also, while there may be more than one correct choice of s_{con} for a given s, we are only interested in finding one.

Let us first consider two common kinds of symmetries: variable and value symmetries. Recall the definitions of these kinds of symmetry: for a given CSP (X, D, C), a variable symmetry is a bijection $X \to X$ and a value symmetry is a bijection $D \to D$. Let us consider only the common branching constraints $x = v$ and $x < v$, where $v \in d(x)$.

For a variable symmetry s, we can represent s_{con}:

$$s_{con}(x = v) \quad \text{as} \quad (s(x) = v) \tag{6.3}$$
$$s_{con}(x < v) \quad \text{as} \quad (s(x) < v) \tag{6.4}$$

That is, for a variable symmetry, it is sufficient to replace the variable x in the branching constraint with its image under the symmetry.

For a value symmetry s, we can represent s_{con}:

$$s_{con}(x = v) \quad \text{as} \quad (x = s(v)) \tag{6.5}$$
$$s_{con}(x < v) \quad \text{as} \quad ? \tag{6.6}$$

In general, it is not clear how to represent $s_{con}(x < v)$ for an arbitrary value symmetry s. However, there is at least one case where there is a simple representation. When s is an inversion (e.g. $s(v) = u - v + l$, where l and u are the lower and upper bounds of the variable), then we can represent:

$$s_{con}(x < v) \quad \text{as} \quad (x > s(v)) \tag{6.7}$$

For example, for a variable x whose domain is $d(x) = \{1..10\}$, the value inversion symmetry maps value v to $10 - v + 1$: 1 to 10, 2 to 9, 3 to 8, etc.

Another common kind of value symmetry is interchangeability, wherein any two values in some set can be interchanged. It is not clear in this case how to define s_{con}. Intuitively, a domain-splitting constraint relies on the values forming a number-line so that the notions of

less-than and greater-than have some meaning. In the case of the value-inversion symmetry described above, the number line is preserved and therefore s_{con} is simple. However, an arbitrary value symmetry (such as transposing 1 with 4) can disrupt the number line, making the derivation of s_{con} complex.

To illustrate the potential complexity, consider a variable x with domain 1..5 and branching constraint $x < 3$. Clearly, $Sol(x < 3) = \{(x = 1), (x = 2)\}$. Let us now consider a value symmetry s that interchanges 3 and 4. Then, $s(Sol(x < 3)) = Sol(x < 3)$ and $s_{con}(x < 3) = (x < 3)$. Now consider the symmetry s' that interchanges 1 and 4. In this case, $s'(Sol(x < 3)) = \{(x = 4), (x = 2)\}$ which is impossible to capture by a primitive constraint.

This property of interchangeable values leads to an interesting consequence for search heuristics: values that are interchangeable do not require domain splitting in order to be searched efficiently. In a sense, domain splitting and dynamic symmetry breaking have the same aim: to reduce the search tree by making one search node act on behalf of many nodes. In domain splitting, it is better to detect failure after branching on $x < 5$ rather than detecting it separately for each of $(x = 1), (x = 2), (x = 3), (x = 4)$. Similarly, for dynamic symmetry breaking, it is better to detect failure (or success) for $x = 1$ once and then ignore the branches $(x = 2), (x = 3), (x = 4)$.

It is known that search heuristics should be chosen to suit the particular problem to be solved. As we have shown, the symmetry of the problem also needs to be taken into account when choosing the branching method. If a problem has interchangeable values, domain splitting is pointless: it is better to instantiate a variable to one (arbitrary) value and ignore the rest. For such a variable, the "least-commitment" philosophy of domain splitting gains nothing.

6.4 Detecting $s_{con}(C_p)$

In order to post symmetry breaking constraints for a symmetry s, the search must be able to detect at a given decision node whether $s_{con}(C_p)$ holds. Note that it is permissible (although undesirable) for the entailment test to claim that $s_{con}(C_p)$ does not hold even if it actually does hold (this is called a false negative), but it is an error for the test to claim that $s_{con}(C_p)$ holds if it actually does not hold (called a false positive). Therefore, incompleteness has two sources: the antecedent not holding at a search node (even though it may do so later), and the entailment test being unable to detect that the antecedent holds.

The entailment test may be performed in different ways; the following subsections describe some alternatives.

6.4.1 Only C_p

One way to detect entailment of $s_{con}(C_p)$ is to consider C_p as a set (rather than a conjunction) of constraints and maintain C_p incrementally as the search progresses. Then, to check if $s_{con}(C_p)$ holds, it is sufficient although not necessary to determine whether $s_{con}(C_p) \subseteq C_p$. The $\subseteq$ relation can be determined by syntactic matching in the case

where C_p and $s_{con}(C_p)$ contain only primitive constraints. This method does not use the state of the constraint store in any way; it relies completely on the branching constraints.

Example 64. Consider the search tree shown in Figure 6.3, for the CSP $(\{x, y\}, \{1..10\}, C)$ where there is the variable symmetry $x \leftrightarrow y$. At the node marked α, we have $C_p = \{(x \leq 5), (y \leq 5)\}$ and $s(C_p) = C_p$; since $s_{con}(C_p) = \{s(x) \leq 5, s(y) \leq 5\} = \{y \leq 5, x \leq 5\}$, we see that $s_{con}(C_p) \subseteq C_p$. □

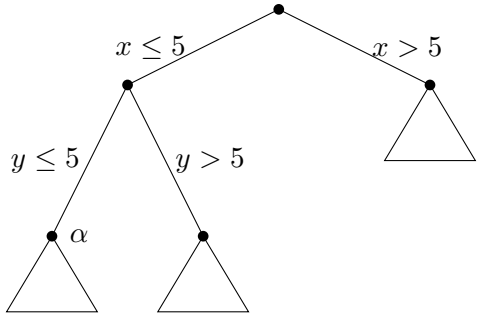


Figure 6.3: At the search node marked α, $s_{con}(C_p)$ holds and is detected, where $s = x \leftrightarrow y$.

This method might result in false negatives whenever inferences due to propagation are not taken into account.

Example 65. Consider the search tree shown in Figure 6.4, for the same CSP as in the previous example. Suppose that propagation on the unspecified constraints removes 5 from the domain of x at the search node marked α; at this point, $C_p = \{(x \leq 5), (y \leq 4)\}$ and therefore $s_{con}(C_p) = \{(y \leq 5), (x \leq 4)\}$. It is clear that $s_{con}(C_p) \not\subseteq C_p$ even though $s_{con}(C_p)$ is in fact entailed: $(y \leq 5)$ is entailed by $(y \leq 4)$, and $(x \leq 4)$ is entailed by the combination of $(x \leq 5)$ and propagation.

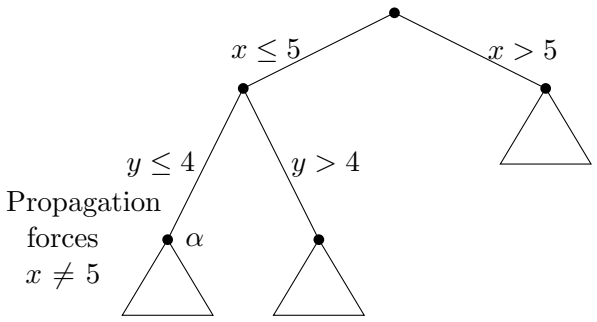


Figure 6.4: Undetected entailment. At the search node marked α, $s_{con}(C_p)$ holds but is not detected, where $s = x \leftrightarrow y$.

□

6.4.2 C_p and the constraint store

The incompleteness (i.e. the number of false negatives) of the previous method can be reduced by examining the constraint store to detect entailment. The constraint store

contains the initial set of problem constraints and those added by the search or as a result of propagation. Typical examples are the domains of variables or variable intervals. For instance, the entailment in the example in Figure 6.4 could be detected by examining the domains of x and y: if $max(x) \leq 4$, then $x \leq 4$ is entailed.

6.4.3 Only the Constraint Store

It is possible, in some cases, to detect that $s_{con}(C_p)$ holds without explicitly keeping track of C_p. This is because it is possible to derive from the constraint store a conjunction G of constraints that entails C_p. Therefore, if it can be shown that $s_{con}(G)$ holds, then it is also true that $s_{con}(C_p)$ holds.

It can also be true that $s_{con}(C_p)$ may hold while $s_{con}(G)$ does not; in such a case, the entailment test would return a false negative.

Example 66. Consider the search tree in Figure 6.5 for the CSP $(\{x, y, ...\}, \{1..10\}, C)$ where there is the variable symmetry $x \leftrightarrow y$. Suppose that propagation on the unspecified constraints forces y to be strictly less than 5 at the search node marked α; at this point, $C_p = \{(x \leq 5)\}$ and therefore $s_{con}(C_p) = \{(y \leq 5)\}$ which is entailed. We derive a conjunction G that is guaranteed to entail C_p by examining how the variables' domains have changed since their initial values: since the upper bound of x is now 5, we add $x \leq 5$ to G, and since the upper bound of y is now 4, we add $y \leq 4$ to G. Now, $s_{con}(G)$ contains $x \leq 4$, which is not entailed by the constraint store. In this case $s_{con}(C_p)$ is entailed but $s_{con}(G)$ is not.

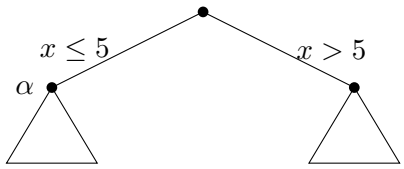


Figure 6.5: Undetected entailment of $s_{con}C_p$ using only the constraint store.

□

Example 67. Let us revisit Example 65; if propagation has reduced the domains of x and y such that the domains are now equal $(\{1, 2, 3, 4\})$, then by using only the constraint store it can be seen that the symmetry s is active. The derived conjunction G is $x \leq 4 \wedge y \leq 4$, using the same method as the previous example. It can be seen that $s_{con}(G) = G$ and is entailed by the constraint store. □

6.5 Effects of Propagation

When checking the entailment of some constraint via the constraint store, the search examines the domain of a particular variable as maintained by the constraint solver. In practice, the domain reported may include values that are logically invalid. For example, a bounds consistent solver may not detect "holes" in a finite domain variable; e.g. given a variable x whose domain is $\{1..10\}$ and constraint $x \neq 5$, the solver may still report the domain of x as $\{1..10\}$.

This incompleteness of constraint propagation means that one must exercise care when using the constraint store to detect entailment. It is easy to contrive a case where reliance on the constraint store coupled with a technically sound solver leads symmetry breaking to prune incorrectly, even for the case of $x = v$ branching constraints.

Example 68. Consider the search tree in Figure 6.6 for the CSP $(\{x, y\}, \{1..10\}, C)$ and assume the variable symmetry s that is $x \leftrightarrow y$. Now, suppose that the solver does not immediately propagate the branching constraint $x = 1$. Therefore, at the node marked α, the domain of x is unchanged (in particular, it is not ground) and the set G is empty. Because $s_{con}(G)$ trivially holds, the search asserts the symmetry breaking constraint $x \neq 1$ on the right branch, which is incorrect: the solution $(1, 2)$ will not be found. Note that the assumption $C_p \subseteq G$ has been broken: $C_p = \{x = 1\}$ but $G = \emptyset$.

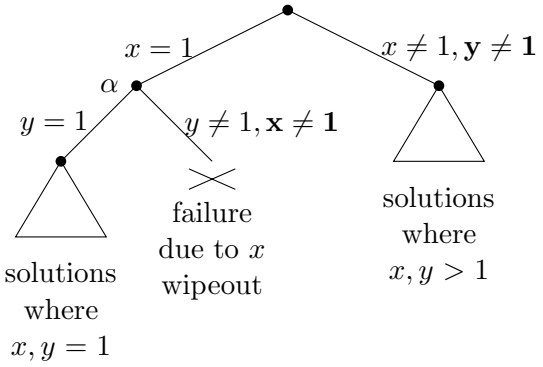


Figure 6.6: Incorrect entailment test due to incomplete propagation.

□

The above example shows that we cannot naively rely on an arbitrary constraint solver to correctly maintain the state of C_p. The problem is that after the $x = 1$ branching constraint is posted, the solver has not immediately updated the domain of x to be the singleton $\{1\}$. As G is constructed only by examining the state of the variables' domains, G does not entail C_p in this case. We need to either modify the search to explicitly track C_p, or impose some minimum requirements on the solver.

In the case of $Var = Val$ style branching constraints, G can be defined as the set of ground variables and their values, i.e. G is the set of all $Var = Val$ such that the domain of x is known to be $\{v\}$. For correctness – that is, to ensure that $G \vdash C_p$ – it is sufficient for the solver to immediately propagate $Var = Val$ constraints. That is, whenever a constraint of the form $Var = Val$ is posted, the variable's domain must appear to be $\{Val\}$ at the following search node.

> **Condition 1**: when a $Var = Val$ constraint is added, at the next search node the domain of Var must appear to be $\{Val\}$.

This condition guarantees that $G \vdash C_p$, because it guarantees that every $(Var = Val) \in C_p$ must then also appear in G at the next search node (and all elements of C_p must have the form $Var = Val$). However, for $x \leq v$ style branching the above requirement

is insufficient; indeed, it has the same problem as the one illustrated in Section 6.3 for SBDS.

Example 69. Consider the search tree shown in Figure 6.7, for the CSP $(\{x, y\}, \{1..10\}, C)$ and assume the symmetry s that is $x \leftrightarrow y$. Assume that at the node marked α, the branching constraint has not been propagated and the domain of x still appears to be $\{1..10\}$. In this case G will be empty and $s_{con}(G)$ will hold trivially, causing the symmetry breaking constraint $x > 5$ to be incorrectly posted on the right child of α. □

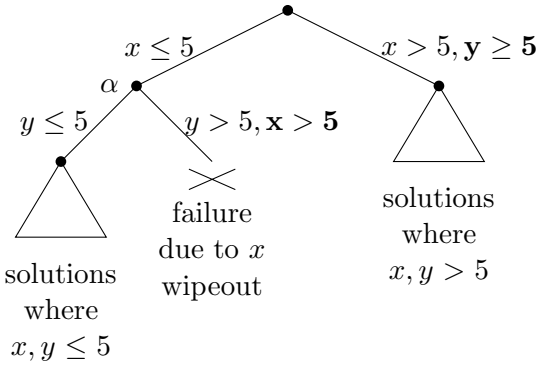


Figure 6.7: Incorrect entailment. Bounds-consistency is required for correctness.

To correctly compute G with *Var* $\leq$ *Val* branching constraints we must examine the bounds of the variable rather than only whether it is ground or not. For *Var* $\leq$ *Val* style branching we can define G as $\{(x \leq v) | x \in X \wedge v = \max(x)\}$. Under this definition G entails C_p only if the upper bound of x as reported by $\max(x)$ is consistent with the constraints posted as part of C_p. Therefore, when a constraint of the form $x \leq v$ is posted, we require the solver to change the upper bound of x to be at most v before the next search node.

> **Condition 2**: when a *Var* $\leq$ *Val* constraint is added, at the next search node the upper bound of *Var* must be at most *Val*.

This condition guarantees that $G \vdash C_p$ as follows. Consider some $(x \leq v) \in C_p$; we show that $G \vdash \{x \leq v\}$. By Condition 2, the upper bound of x is u, where $u \leq v$. Therefore, $x \leq u$ is in G. Since $x \leq u \Rightarrow x \leq v$, it must be that $G \vdash \{x \leq v\}$.

6.6 Domain Splitting and C_p

With domain splitting and first-fail (minimum domain size) variable orderings, the leftmost branch of a search tree typically resembles Figure 6.8. A variable – in this example, the variable x whose domain is 1..10 – is chosen at the root and its domain is split. On the right branch, a symmetry breaking constraint is posted to deal with the x-y symmetry. On the left branch, x is likely to have the smallest domain and is chosen again to have its domain split. However, at the node marked α, the variables x and y are no longer symmetric (because $s_{con}(C_p)$ does not hold) and therefore on α's right branch the search

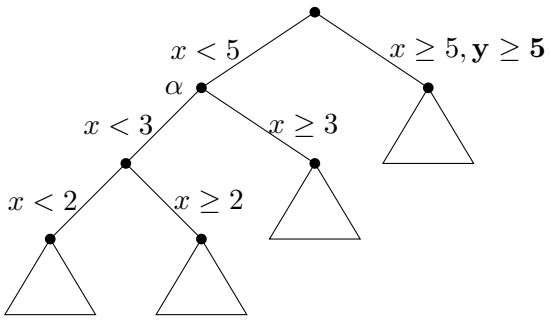


Figure 6.8: Typical search tree with domain splitting and first-fail heuristic. One variable (here x with domain 1..10) is instantiated before any other variable is considered.

does *not* post the symmetry breaking constraint $y \geq 3$. This leads to very little symmetry breaking and a large search tree.

However, it is correct to assert $y \geq 3$ on α's right branch. Suppose that the search had chosen to branch on $x < 3$ and $x \geq 3$ as the root of the search tree. In this case it is clear that posting the symmetry breaking constraint would be correct. The only difference in Figure 6.8 is that $C_p = \{x < 5\}$ instead of $C_p = \emptyset$. However, within α's left subtree the constraint $x < 5$ is irrelevant because it is subsumed by the constraint $x < 3$ – it is as if $x < 5$ had never been posted. Therefore, the subsumed constraint $x < 5$ should have no bearing on our symmetry breaking at α.

In general, at search node v, a constraint $p \in C_p$ is subsumed if $(C_p \cup \{c\}) \setminus \{p\} \vdash p$. Let us denote the set of subsumed constraints by $S \subseteq C_p$. Given the above discussion, now instead of checking for $s_{con}(C_p)$ before posting a symmetry breaking constraint, we need check only for $s_{con}(C_p \setminus S)$.

In our experiments we have found that this reduced checking leads to significantly improved behaviour when combining dynamic symmetry breaking with domain splitting and most variable orderings.

Branch	Var	Time (s)			Failures			Solutions		
		No	St	Dy	No	St	Dy	No	St	Dy
inst	lex	467.88	1.66	4.35	123696668	35831	89246	480	1	2
	anti	4610.14	57.51	4.32	123696668	844418	89246	480	1	2
	mind	5044.27	1.46	5.01	136335934	31728	103601	480	1	2
	maxd	1359.99	13.72	6.08	22496066	204632	97902	480	1	2
split	lex	3839.25	1.34	3.38	85564622	25936	60247	480	1	2
	anti	3764.92	27.68	3.31	85564622	393801	60247	480	1	2
	mind	4159.57	1.40	4.02	92959204	27538	73180	480	1	2
	maxd	440.76	1.30	4.16	9002656	25010	82231	480	1	4

Table 6.1: Results for finding all solutions for the graceful graph problem $K_5 \times P_2$.

6.7 Experiments

We have attempted to evaluate the behaviour of a static symmetry breaking method against that of LDSB with domain splitting under different search heuristics. We have implemented domain splitting for LDSB using the second of the entailment tests described above – explicitly maintaining C_p and testing entailment using the constraint store. We used Gecode (Gecode Team, 2006) to run the experiments.

We have used the $K_n \times P_2$ Graceful Graph problem (see Section A.9) as a benchmark. This problem can be modelled in several ways (see (Puget and Smith, 2006)); we have chosen the simplest model with one decision variable per graph vertex. There are three forms of symmetry: (1) a complement symmetry among the values, where value i is mapped to $q - i$ (q is the number of edges in the graph), (2) the two K_n cliques can be interchanged, and (3) the vertices in both cliques can be permuted as long as both cliques are acted upon identically.

The symmetry in this problem can be elegantly broken for graphs of the with static symmetry breaking constraints (Smith, 2006). The static constraints impose an ordering on the vertices of one clique by constraining the first vertex's label to be 0, and ordering the other vertices in the first clique arbitrarily. In addition, there is a constraint to ensure that the vertex labelled 24 ($q - 1$) is adjacent to the first vertex (labelled 0)

Table 6.1 compares three different approaches to finding all solutions to the $K_5 \times P_2$ problem: static symmetry breaking (St) with the static constraints described above, our extended LDSB (Dy) with the three symmetries described above, and no symmetry breaking (No). In the branching method column, "inst" denotes $x = v$ style branching, and "split" denotes $x \leq v$ style. For variable order, "lex" assigns domain values to each variable in increasing lexicographical order until it is ground, "anti is the same as "lex" but in reverse order, and "mind and "maxd branch on the variable with the smallest or largest domain size, respectively.

The results show the effects of the interaction between variable ordering, branching constraints and symmetry breaking. It can be seen that for all symmetry breaking methods and variable orderings, domain splitting is uniformly more efficient. When the static symmetry breaking constraints cooperate with the variable ordering, the search tree is small and the search finishes quickly. However, when the variable ordering conflicts with the constraints, the search tree size and time increases dramatically. On the other hand, the variable ordering has very little effect on the efficiency of the dynamic symmetry breaking method, which seems to be quite consistent: neither as fast nor as slow as the extremes of the static method.

6.8 Conclusions

We have shown how to extend an implementation of our dynamic symmetry breaking method LDSB to cooperate with the common search technique of domain splitting. The very preliminary experimental results show that the execution time of the resulting dynamic symmetry breaking method is more predictable in its interaction with variable ordering heuristics than static symmetry breaking, which can be faster but can also perform badly if it conflicts with the variable ordering. This suggests that dynamic symmetry

breaking is less sensitive to the variable ordering and may be preferred if the static symmetry breaking constraints cannot be designed to agree with the variable ordering.

In addition, we have shown that for certain kinds of symmetry breaking the constraint solver must obey some simple conditions in order for symmetry breaking to be correct. These conditions ensure that the symmetry breaking method can rely on the domains of the variables as reported by the solver.

It is clear that symmetry breaking is a vital component of a successful constraint programming technique. Equally, the choice of the appropriate variable ordering and kind of branching constraints is crucial to search performance. There has been little exploration of how these two techniques for improving search behave when they are combined. In this chapter we have discussed a preliminary exploration of how the domain-splitting heuristic and different variable orderings interact with dynamic and static symmetry breaking. However, there is much scope for further work in this area.

Here we have covered $Var = Val$ and $Var \leq Val$ branching constraints. Although these kinds of branchings are very common, they are by no means the only possibilities; for some problems it may be useful to branch using binary constraints to state the relationship between two variables, or perhaps even more complex constraints. It is not clear how dynamic symmetry breaking might interact with such a search. Also, we have considered only variable and value symmetries independently and not discussed how a s_{con} might be defined for variable-value symmetries. This is the subject of further research.

Chapter 7

Conclusion

The study of symmetry in constraint programming is a challenging and productive area of research. It has been shown that exploiting symmetry can lead to considerable improvements in performance when searching for solutions to combinatorial problems. In this thesis we have provided improvements to all parts of the use of symmetry in constraint programming, from the detection stage through to the exploitation stage. Let us now discuss in detail the contributions that have been made, and the future work that they give rise to.

Instance Symmetry Detection. There has been a significant amount of work done in the detection of symmetries in constraint program instances. We have examined in detail one of the most promising methods of automatically detecting symmetries in constraint satisfaction problems, that presented by Puget (2005a). This method is quite efficient and in its most general form is able to detect all kinds of constraint symmetry in a problem, not merely variable- or value-symmetries. However, the method was not formally defined and its informal definition is not complete. As a result, it is not clear how it is applied to certain CSPs. Moreover, no attempt was made at proving its correctness and, in fact, we have shown that in some cases it produces incorrect results. In addition, much of its benefit appears to come from its special treatment of the all-different global constraint. While the savings in graph size are substantial, introducing custom-designed representations for constraints risks losing some of the symmetry-detecting power of the method since symmetries arising from the conjunction of different kinds of constraint can no longer be captured. It is also not clear how such special representations could be extended to other kinds of global constraint. In this context, our contributions to the detection of symmetry in problem instances is as follows:

- We have identified CSPs for which the previous method gives incorrect results, and provided modifications that are sufficient to ensure that the resulting method is correct.
- We have formally defined new forms of graph construction that can be used to represent a constraint satisfaction problem and its symmetries. We describe the disallowed assignments graph and allowed assignments graph and also a form, the full assignments graph, that combines the two to make a more compact representation. We discuss in depth how our new constructions reflect the symmetries of the original

problem and prove that the symmetries found in these graphs do indeed correspond to symmetries of the problem.

- We have described two ways of reducing the size of the graphs representing the problems. First, we show that enforcing arc-consistency on the problem preserves correctness and keeps most, though not all, of the symmetries of the original problem. This gives dramatic size improvements in some cases, such as for set variables with cardinality constraints. Second, we show that reducing the arity of the constraints leads to a smaller graph with possibly more symmetries – a transformation that runs counter to the customary approach of constraint programming, which is to use larger, more complex constraints.

- We have implemented our graph construction and used it to detect the symmetries of a range of benchmark problems. The results show that our method finds all of the symmetries that Puget's best graph construction can find, and does not require specially crafted representations of global constraints to do so. Our method has much faster running times for some problems but sometimes slower running times due to the special all-different representation.

The development of our new graph construction for symmetry detection raises some topics for future research. Puget's method of graph construction leads to compact graphs for certain kinds of problem, thanks to the use of a representation specifically designed for the all-different constraint. It remains to be seen whether this approach to making smaller graphs can be extended effectively to other kinds of constraints, or indeed whether this is a desirable trade-off against the amount of symmetries found.

Our method is not without limitations: despite the improvements brought by our method of graph construction, we are still limited by the size of the problem to be analysed. As the problem grows in the number of variables, the sizes of their domains and the number of constraints, the graph's size grows quickly. As a consequence, our method cannot be used on very large problems, where we might desire to know the symmetries because the savings gained in exploiting them would be substantial. However, this limitation is not a concern because our interest lies in detecting the symmetries of problem models, where we use our instance symmetry detection method only on small instances.

Model Symmetry Detection. In contrast to the work done for individual problems, there has been little research in finding symmetries of constraint satisfaction models. The task is very difficult, owing to the lack of standard notation for problem models and the inherent hardness of program analysis. The only two methods we know that focused on models are limited in their use either by requiring custom application to each problem, or by being dependent on the syntax used to express the problem: only the symmetries of global constraints were considered. The latter kind can be seen as an abstract interpretation method (Cousot and Cousot, 1977) where a loss of accuracy occurs in the abstraction of symmetries (only the symmetries inherent in global constraints are considered) and when conjoining them abstractly. Our contributions to the detection of symmetries in problem models is as follows:

- We have presented a radically novel approach to automatically finding symmetries in constraint satisfaction models. Our approach exploits the effectiveness of our

graph-based instance detection method discussed earlier while avoiding its main drawback of inefficiency in large problems. The foundation of the approach is to construct several distinct small instances of the model, to find the symmetries of those instances, and then to generalise those symmetries back to the model. Once generalised, these symmetries are filtered to select likely candidates to be symmetries of the model. The final step is to confirm that each candidate is indeed a symmetry of every instance of the model, or to discard that candidate.

- We have provided an implementation of each component of our new framework. Our implementation considers problems whose parameters are a vector of integers, and constructs instances of the model by generating parameters from a base parameter vector. For each of these instances, our implementation uses the symmetry detection method described in Chapter 3 to find the generating set of symmetries of the problem. The next step is to generalise the symmetries, which our implementation does by matching each symmetry against a set of patterns. The patterns include some of the symmetries that occur most commonly in constraint problems, such as row- and column-swaps and reflections in matrices of variables. After matching against patterns, we take the intersection of the generalised symmetries of all instances, using a computational group theory package to avoid problems caused by the non-uniqueness of generating sets.

- We have tested our implementation on several varied benchmark problems and found that it performs very well. Indeed, it finds almost all of the symmetries of the problems, and does so without having to examine large problem instances.

- We have presented two methods for automatically proving that candidate symmetries are symmetries of a model. These methods, described informally by way of examples, are able to prove the correctness of some kinds of symmetry, and suggest promising avenues that may lead to more general solutions.

- We have described how the framework, originally designed for symmetry, can be applied to the detection of other properties. In particular, we have discussed how the framework can be applied to the task of finding opportunities for caching the results of sub-problems during constraint satisfaction search. The development and implementation of the framework applied to this problem is future work whose fulfilment would be extremely useful to constraint programming.

As our approach to detecting symmetries of constraint models is novel, there remain many open questions. One of prime importance is the applicability of the framework to models where the parameters are not vectors of integers, such as where the data may be a graph or some other structured information. Indeed, it is a limitation of the method that a symmetry that is contingent on the data itself, such as a symmetry that exists only in the data, cannot be found. In addition, the task of proving that candidate symmetries apply to the model remains a difficult one. Although the framework and our implementation are useful even without a complete method for this step, a fully automatic way to perform this task would make its use more convenient for practitioners.

Symmetry Breaking. The main purpose of detecting symmetries in constraint programming is to use them to improve the performance of the search for solutions to a problem. Symmetry breaking has proved vital for the practical resolution of some problems, where the symmetries that exist in the problems cause a very large amount of redundant work in a naive search. However, they have also proved to be difficult to use, either due to unpredictable performance or to the high expertise required from the user. In the existing literature there is no static or dynamic method that is easy for a practitioner to use and that gives good, consistent results in general.

- We have developed a symmetry breaking method called Lightweight Dynamic Symmetry Breaking, or LDSB. The strengths of LDSB are that it adds little overhead to the search, it imposes a small burden on the user, and it often significantly improves search performance. LDSB achieves this by focusing on symmetries that are common in practice and that can be represented simply and manipulated efficiently. LDSB is a formalisation of the shortcut SBDS method, which aims to increase efficiency at the expense of completeness. Importantly, there is no need to use any computational group theory software – such as GAP, which is commonly used in the dynamic symmetry breaking in platforms such as ECL^{i}PS^e– nor is there any need for the user to input the whole symmetry group, which can in practical problems number millions of symmetries. This is not needed in LDSB because composition of variable and value symmetries is built-in, so most of the symmetries resulting from composition are automatically broken.

- We have provided two publicly-available implementations of LDSB for the popular constraint programming platforms ECL^{i}PS^e and Gecode. Both of these implementations are to be distributed with these platforms, so that they can be used very easily by constraint programmers. We expect that the distribution of symmetry breaking methods with popular constraint systems will stimulate more interest and improvements in the area. Our experiments with LDSB show that despite its simplicity it is at worst competitive with other dynamic symmetry breaking methods and often surpasses them markedly in performance. In particular, LDSB's performance is consistent and predictable while other methods vary greatly in their behaviour.

LDSB has two main limitations. First, it cannot represent variable-value symmetries, and second, it is not complete, i.e. it does not guarantee that all symmetries will be fully broken. We believe, and our experiments suggest, that this trade-off is worthwhile. It remains to be seen what further improvements could be made to LDSB. Just as reducing the kinds of symmetry handled has led to performance improvements, it is likely that other shortcuts – such as breaking symmetries only at the top of the search tree – may lead to increased efficiency.

Usability of Symmetry Breaking. A symmetry breaking method does not act in a vacuum; it must interact with a constraint solver and a search algorithm. The way in which symmetry breaking cooperates (or fails to cooperate) with these components can have a dramatic effect on the efficiency of search.

- We have shown that an apparently straight-forward extension of the SBDS symmetry breaking method to handle $x < v$ style branching constraints is in fact incorrect.

- We have discussed the implications of extending LDSB (and SBDS, on which it is based) to use branching constraints of the form $x < v$. We also show how the symmetries present in a problem can help to guide the choice of branching constraint.
- We have analysed the interaction between LDSB and the constraint solver with which it is used. As LDSB uses the domains of the variables in a problem to make decisions, it relies on the ability of the constraint solver to report those domains correctly. We provide two minimum conditions that a solver must meet in order for it to be used correctly with LDSB, both for $x = v$ and $x < v$ branching constraints.
- We have shown, with some preliminary experiments, how the choice of static or dynamic symmetry breaking, the choice of variable ordering in search and the choice of branching constraint interact. Our dynamic symmetry breaking method, LDSB, appears to be much less sensitive to changes in variable ordering that can cause a large variation in running time for a static symmetry breaking method.

Although we have covered some of the most common cases, there is much left to be explored in these issues. It is not known how dynamic symmetry breaking could be applied to non-unary branching constraints. Similarly, we have discussed variable symmetries and value symmetries, but leave the consideration of variable-value symmetries to the future. Finally, there is great scope for further experimental comparison and even composition of dynamic and static symmetry breaking methods under search algorithms using dynamic and static variable and value orderings.

In summary, in this thesis we have provided a single integrated and implemented system for all phases of symmetry handling in constraint programming, from automatic detection through to automatic symmetry breaking in search.

Appendix A

Benchmark Problems

This appendix provides details regarding the benchmark problems used in the thesis. For each benchmark, it provides a description of the problem together with a Zinc (Garcia de la Banda et al., 2006) model (note that some of these models are example models distributed with Zinc). The Zinc models are intended to be succinct specifications of the problem rather than the most efficient model.

A.1 N-queens

The *N-Queens* problem is to place N chess queens on an $N \times N$ chessboard in such a way that no queen attacks another queen. That is, no two queens may face each other along a row, column or diagonal.

There are two commonly-used model of the problem: one with an integer variable per queen, and the other with a Boolean variable per square.

A.1.1 Integer model

In any solution to the N-queens problem there must be exactly one queen per column. We can solve the problem by determining, for each column, in which row the queen is placed. The constraints ensure that each row is taken by exactly one queen, and that each diagonal by at most one queen. (The model show was taken from the G12 project example (G12 Project, 2009).)

```
%% One queen per column.
%% An array q of size n of variables with domain 1..n.
array [1..n] of var 1..n: q;

constraint
        %% For each pair (i,j) of queens.
        forall (i in 1..n, j in i + 1..n) (
                %% Queen i doesn't attack queen j.
                q[i]      != q[j]
        ∧       q[i] + i != q[j] + j
```

```
    ⋀       q[i] - i != q[j] - j
    );
```

A.1.2 Boolean model

In the Boolean model, each square on the board is represented by a Boolean variable which is 1 if a queen occupies that square and 0 otherwise. Constraints ensure that each row and column has exactly one queen, and each diagonal has at most one queen. (The model show was taken from the G12 project example (G12 Project, 2009).)

```
type rg = 1 .. n;
%% 2d array of Boolean variables.
array [rg, rg] of var 0 .. 1: q;

constraint forall (i in rg) (
   % Exactly one queen per row.
   ( sum (j in rg)                    (q[i, j])      = 1 )
   % Exactly one queen per column.
⋀( sum (j in rg)                    (q[j, i])      = 1 )
   % At most one queen per diagonal.
⋀( sum (j, k in rg where j−k = i−1) (q[j, k])      ≤1 )
⋀( sum (j, k in rg where j−k = 1−i) (q[j, k])      ≤1 )
⋀( sum (j, k in rg where j−k = i−1) (q[n+1−j, k]) ≤1 )
⋀( sum (j, k in rg where j−k = 1−i) (q[n+1−j, k]) ≤1 )
);
```

A.2 Social Golfers

The *Social Golfers* problem requires a weekly schedule to be created for a group of golfers. There are $G \times P$ golfers, who are to be arranged every week in G groups each of size P, over a period of some weeks. Two golfers may not play in the same group more than once; that is, a player must play with different people every week. (The model show was taken from the G12 project example (G12 Project, 2009).)

```
% "Weeks" is number of weeks.
% "Groups" is number of groups per week.
% "GroupSize" is the size of each group.
% "Players" is the set of players
%     (the numbers 1..Groups*GroupSize).

% 2D array of size (Weeks x Groups) of set variables, each
% of whose elements are drawn from the set of Players.
array[1..Weeks,1..Groups] of var set of Players: group;

% Constrain an array of set variables such that the maximum
```

```
% overlap between any two sets is n. The elements of the sets
% are of arbitrary type (the type variable $E).
predicate maxOverlap(array[int] of var set of $E: sets, int: n) =
    forall(i,j in 1..length(sets) where i < j) (
        card(sets[i] intersect sets[j]) ≤n
    );

% Groups are always the same size.
constraint
    forall (i in 1..Weeks, j in 1..Groups) (
        card(group[i,j]) == GroupSize
    );

% No golfer plays in more than one group each week.
constraint
    forall (i in 1..Weeks) (
        maxOverlap([group[i,j] | j in 1..Groups], 0)
    );

% Groups change each week, ie. no golfer plays in the same group
% as any other golfer twice.
constraint
    maxOverlap([group[i,j] | i in 1..Weeks, j in 1..Groups], 1);
```

A.3 Golomb Ruler

A *Golomb Ruler* is a set of m integer marks on a ruler $\{a_1, a_2, \ldots, a_m\}$ in ascending order such that all differences between marks $a_i - a_j$ are different. The first mark a_1 is taken to be 0 and the length of the ruler is a_m. The task is to minimise the length of the ruler for a given number of marks. (The model show was taken from the G12 project example (G12 Project, 2009).)

```
int: n = m*m;

% Array of marks.
array [1..m] of var 0..n: mark;

% Array of differences, one per pair of marks.
array[int] of var 0..n: differences =
   [ mark[j] - mark[i] | i in 1..m, j in i+1..m ];

% First mark must be zero, the marks must be increasing,
% and the differences must be distinct.
constraint mark[1] = 0;
constraint forall(i in 1..m-1) (mark[i] < mark[i+1]);
```

```
constraint all_different ( differences );
```

A.4 N × N Queens

The $N \times N$ *Queens* problem is to find N simultaneous, non-overlapping solutions to the N-Queens problem. It can also be viewed as a graph colouring problem. Each square of the $N \times N$ chessboard is a vertex, and there is an edge (u, v) if a chess queen could move from u to v in a single move on an otherwise empty board. The task is to colour this graph with N colours.

```
% 2d matrix of integers.
array [1..n, 1..n] of var 1..n : board;

% All values in a row must be different.
constraint forall(i in 1..n) ( all_different ([board[i,j] | j in 1..n]));
% All values in a column must be different.
constraint forall(j in 1..n) ( all_different ([board[i,j] | i in 1..n]));

% All values in any diagonal must be different, in each direction.
constraint forall(x in -n..n)
    ( all_different ([board[i,j] | i,j in 1..n where i-j == x]));
constraint forall(x in 1..2*n)
    ( all_different ([board[i,j] | i,j in 1..n where i+j == x]));
```

A.5 Balanced Incomplete Block Design

A *Balanced Incomplete Block Design* is an arrangement of v objects into b blocks such that each block contains exactly k objects, each object occurs in exactly r blocks, and every two objects occur together in exactly λ blocks. The BIBD generation problem is to generate such arrangements.

The model has a $b \times v$ matrix of Boolean variables, where the variable at index (i, j) is 1 if and only if block i contains object j. (The model show was taken from the G12 project example (G12 Project, 2009).)

```
type Blocks = 1..b;
type Objects = 1..v;

% 2d array of Booleans; bibd[b,v] is 1 if object v is
% in block v.
array[Blocks, Objects] of var bool: bibd;

% Must be k objects per block.
constraint
    forall (b in Blocks)
```

```
        ( sum (v in Objects) (bool2int(bibd[b, v])) = k );

% Each object must occur in r blocks.
constraint
    forall (v in Objects)
        ( sum (b in Blocks) (bool2int(bibd[b, v])) = r );

% Each pair of objects occurs together in lambda blocks.
constraint
    forall (v1,v2 in Objects where v1 < v2)
        ( sum (b in Blocks) (bool2int(bibd[b, v1] /\ bibd[b, v2])) = lambda );
```

A.6 Steiner Triples

The *Steiner Triple* problem requires a set of N objects to be arranged into triples such that every two objects appear together in exactly one triple. For example, an arrangement for the set $\{1, 2, 3, 4\}$ is $\{(1, 2, 3), (1, 4, 5), (1, 6, 7), (2, 4, 6), (2, 5, 7), (3, 4, 7), (3, 5, 6)\}$.

```
m = n*(n-1) div 6;

% Each triple is a set variable.
array[1..m] of var set of 1..n : triples;

% Each set variable must be a triple (cardinality 3).
constraint forall(i in 1..m) (card(triples[i]) == 3);

% Each pair of sets must have at most one value
% in common.
constraint forall(i,j in 1..m where i<j)
  (card(triples[i] intersect triples[j]) <= 1);
```

A.7 Steel Mill Slab Design

The *Steel Mill Slab Design* problem requires orders of steel to be assigned to slabs. Each order has a weight and a colour, and each slab has a capacity chosen from a specified set. The sum of the weights of the orders assigned to a slab must not exceed the slab's capacity, and orders using at most two different colours may be assigned to one slab.

```
int : norders;

type range = 1..norders;

% Permitted slab capacities.
array[int] of int : capacities;
```

```
% An order has a weight and a colour.
type order = record(int : weight, int : colour);

% The orders.
array[range] of order : orders;
% The assignment of orders to slabs.
array[range] of var range : x;

% The load on each slab.
array[range] of var 0..maxcapacity : load;

% The largest permitted capacity.
int : maxcapacity = max(capacities);

% The load on a slab is equal to the sum of the weights
% of the orders assigned to that slab.
constraint
  forall (i in range)
    (load[i] == sum (j in range) (bool2int(x[j] == i)*orders[j].weight));

% No slab may exceed the maximum permitted capacity.
constraint
  forall (i in range)
    (load[i] <= maxcapacity);

% The largest colour in any order.
int : maxcolour = max([orders[i].colour | i in range]);

array[range, 1..maxcolour] of var int : colours_used;
array[range, 1..maxcolour] of var bool : colours_used_bool;

array[1..maxcolour] of set of range : orders_by_colour =
  [ c : { i | i in range where orders[i].colour == c } | c in 1..maxcolour ];

% Ensure that colours_used[s,c] is 1 if slab s contains an
% order using colour c.
constraint
  forall (s in range,c in 1..maxcolour)
    (count([x[i] | i in orders_by_colour[c]], s, colours_used[s,c])
     /\
     colours_used_bool[s,c] == (colours_used[s,c] > 0));

% The number of different colours used in any slab is at most 2.
constraint
```

```
  forall (s in range)
    (sum(c in 1..maxcolour) (bool2int(colours_used_bool[s,c])) ≤ 2);

% The loss of a slab is the amount of unused capacity.
array[0..maxcapacity] of int : loss =
  [ i : min([j | j in capacities where j ≥i]) − i | i in 0..maxcapacity ];

array[range] of var int : waste;

constraint
  forall (i in range)
    (waste[i] == loss[load[i ]]);

% The total loss is the value to minimised.
var int : totalloss = sum(waste);
```

A.8 Latin Square

A *Latin Square* is an $N \times N$ matrix where the symbol in each cell is drawn from a set of N symbols, and no symbol appears twice in any row or twice in any column.

```
int : n;
type range = 1..n;

array[range,range] of var range : square;

constraint
  % The values in each row/column must be distinct.
  forall (i in range)
    ( all_different ([square[i,j] | j in range])
     ∧
      all_different ([square[j,i] | j in range]));
```

A.9 Graceful Graph Labelling

The *graceful graph labelling* problem is to assign an integer value to each vertex of a graph so that all vertex labels are distinct, and so that all edge labels are distinct. The label on an edge (u, v) must be the absolute difference of the label on u and the label on v, and the vertex labels must be drawn from the set $\{0..q\}$, where q is the number of edges in the graph.

```
% The number of graph vertices.
int : nodes;
type node = 1..nodes;
```

```
% The set of graph edges.
type edge = tuple(node,node);
set of edge : edges;

% The labels assigned to graph vertices.
type label = 0..length(edges);
array[node] of var label : m;

% All labels must be distinct.
constraint all_different (m);

% The edge labels.
type edgelabel = 1..length(edges);
array[edgelabel] of var edgelabel : differences;

% The label on an edge is the absolute difference of
% the labels on its endpoints.
constraint
  forall(i in 1..length(edges))
    (abs(m[edges[i].1] - m[edges[i].2]) == differences[i]);

% All edge labels must be distinct.
constraint all_different ( differences );
```

A.10 Concert Hall Scheduling

The *concert hall scheduling* problem is to determine which subset of offers to accept in order to maximise profit. A set of offers is made to rent a concert hall, each with a start time, an end time and a price. The offers should be accepted so that the profit (the sum of the prices of accepted offers) is a maximum, under the constraint that two offers that overlap in time cannot both be accepted.

```
% An offer has a start time, and end time, and a price.
type offer = record(int : start, int : end, int : price);
array[int] of offer : offers;

type r = 1..length( offers );

% x[i] is true if offer i is accepted.
array[r] of var bool : x;

% Do offers i and j overlap in time?
function bool : overlap(int : i, int : j) =
  offers[i].start ≤ offers[j].end ⋀
```

```
  offers [j]. start ≤ offers [i]. end ;

% Any two overlapping offers are mutually exclusive.
constraint
  forall (i,j in r where i < j ⋀overlap(i,j))
    ((x[i] ⋀x[j]) == false);

% The profit is the sum of the prices of accepted offers,
% and is to be maximised.
var int : profit = sum(i in r) (bool2int(x[i]) * offers [i]. price );
```

A.11 Graph Colouring

The *graph colouring* problem is to colour the vertices of a graph using a finite number of colours such that the endpoints of every edge have different colours.

```
% Graph vertices.
int : nodes;
type node = 1..nodes;

% Graph edges.
type edge = tuple(node,node);
set of edge : edges;

% Number of permitted colours.
int : colours;
type colour = 1..colours;

% Assignment of colours to edges.
array[node] of var colour : x;

% The endpoints of each edge must have different colours.
constraint
  forall(i in 1..length(edges))
    (x[edges[i].1] != x[edges[i].2]);
```

Vita

Publications arising from this thesis include:

C. Mears, M. Garcia de la Banda, M. Wallace On Implementing Symmetry Detection. In *The 6th International Workshop on Symmetry in Constraint Satisfaction Problems.* 2006.

B. Demoen, M. Garcia de la Banda, C. Mears, M. Wallace A Novel Approach For Detecting Symmetries in CSP Models. In *The 7th International Workshop on Symmetry in Constraint Satisfaction Problems.* 2007.

C. Mears, M. Garcia de la Banda, M. Wallace, B. Demoen A Novel Approach For Detecting Symmetries in CSP Models. In *The 5th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems.* 2008.

C. Mears, M. Garcia de la Banda, B. Demoen, M. Wallace Lightweight Dynamic Symmetry Breaking. In *The 8th International Workshop on Symmetry in Constraint Satisfaction Problems.* 2008.

C. Mears, M. Garcia de la Banda, M. Wallace, B. Demoen Symmetry Breaking and Branching Constraints. In *The 8th International Workshop on Symmetry in Constraint Satisfaction Problems.* 2008.

C. Mears, M. Garcia de la Banda, M. Wallace On Implementing Symmetry Detection. *Constraints*, volume 14. 2009.

Permanent Address: Clayton School of Information Technology
Monash University
Australia

This thesis was typeset with LaTeX 2$_\varepsilon$[1] by the author.

[1] LaTeX 2$_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Glenn Maughan and modified by Dean Thompson and David Squire of Monash University.

References

K.R. Apt and M.G. Wallace. *Constraint Logic programming using ECLiPSe*. Cambridge University Press, 2006. ISBN 0521866286.

R. Backofen and S. Will. Excluding symmetries in constraint-based search. In *Principles and Practice of Constraint Programming - CP 1999*, 1999.

N. Beldiceanu, M. Carlsson, S. Demassey, and T. Petit. Global constraint catalogue: Past, present and future. *Constraints*, 12(1), 2007.

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

A. Bossi and Y. Deville. Special issue: synthesis, transformation and analysis of logic programs. *The Journal of Logic Programming*, 39(1-3), 1999.

J. Charnley, S. Colton, and I. Miguel. Automatic generation of implied constraints. In *ECAI 2006: 17th European Conference on Artificial Intelligence*, 2006.

D. Cohen, P. Jeavons, C. Jefferson, K.E. Petrie, and B.M. Smith. Symmetry definitions for constraint satisfaction problems. In *Principles and Practice of Constraint Programming - CP 2005*, 2005.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 1977.

J. Crawford, M.L. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *KR'96: Principles of Knowledge Representation and Reasoning*, 1996.

P.T. Darga, M.H. Liffiton, K.A. Sakallah, and I.L. Markov. Exploiting structure in symmetry generation for CNF. In *Proceedings of the 41st Design Automation Conference*, 2004.

T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *Principles and Practice of Constraint Programming - CP 2015*, 2001.

P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In *Principles and Practice of Constraint Programming - CP 2002*, 2002a.

P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Matrix modelling: Exploiting common patterns in constraint programming. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems*, 2002b.

P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. *Logic Based Program Synthesis and Transformation, 13th International Symposium LOPSTR 2003 Revised Selected Papers*, 2004.

F. Focacci and M. Milano. Global cut framework for removing symmetries. In *Principles and Practice of Constraint Programming - CP 2001*, 2001.

E.C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11), 1978.

E.C. Freuder. Eliminating interchangeable values in constraint satisfaction problems. In *Proceedings of The Ninth National Conference on Artificial Intelligence (AAAI-91)*, 1991.

A.M. Frisch, C. Jefferson, and I. Miguel. Constraints for breaking more row and column symmetries. In *Principles and Practice of Constraint Programming - CP 2003*, 2003a.

A.M. Frisch, I. Miguel, and T. Walsh. CGRASS: A system for transforming constraint satisfaction problems. In *Recent Advances in Constraints, Joint ERCIM/CologNet International Workshop on Constraint Solving and Constraint Logic Programming*, 2003b.

A.M. Frisch, M. Grum, C. Jefferson, B. Martinez Hernandez, and I. Miguel. The design of ESSENCE: A constraint language for specifying combinatorial problems. In *International Joint Conference on Artificial Intelligence*, 2007.

G12 Project. G12 constraint programming platform, 2009. See `http://www.nicta.com.au/`.

M. Garcia de la Banda and P.J. Stuckey. Dynamic programming to minimize the maximum number of open stacks. *Informs Journal on Computing*, 19(4), 2007.

M. Garcia de la Banda, M. Hermengildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global analysis of constraint logic programs. *ACM Transactions on Programming Languages and Systems*, 1996.

M. Garcia de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP languages. *ACM Transactions on Programming Langugages and Systems*, 22(2), 2000.

Maria Garcia de la Banda, Kim Marriott, Reza Rafeh, and Mark Wallace. The modelling language Zinc. In *Principles and Practice of Constraint Programming - CP 2006*, 2006.

A. Gargani and P. Refalo. An efficient model and strategy for the steel mill slab design problem. In *Principles and Practice of Constraint Programming - CP 2007*, 2007.

Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

S. Genaim and M. Codish. Inferring termination conditions for logic programs using backwards analysis. *Theory and Practice of Logic Programming*, 5, 2005.

I.P. Gent and B.M. Smith. Symmetry breaking in constraint programming. In *ECAI 2000: 14th European Conference on Artificial Intelligence*, 2000.

I.P. Gent and T. Walsh. CSPLib: a benchmark library for constraints. Technical report, Technical report APES-09-1999, 1999. Available from http://csplib.cs.strath.ac.uk/.

I.P. Gent, W. Harvey, and T. Kelsey. Groups and constraints: Symmetry breaking during search. In *Principles and Practice of Constraint Programming - CP 2002*, 2002.

I.P. Gent, W. Harvey, T. Kelsey, and S. Linton. Generic SBDD using computational group theory. In *Principles and Practice of Constraint Programming - CP 2003*, 2003.

I.P. Gent, T. Kelsey, S. Linton, I. McDonald, I. Miguel, and B.M. Smith. Conditional symmetry breaking. In *Principles and Practice of Constraint Programming - CP 2005*, 2005.

I.P. Gent, C. Jefferson, and I. Miguel. MINION: A fast, scalable, constraint solver. In *ECAI 2006: 17th European Conference on Artificial Intelligence*, 2006.

The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4.9*, 2006.

D.S. Heller, A. Panda, M. Sellmann, and J. Yip. Model restarts for structural symmetry breaking. In *Principles and Practice of Constraint Programming - CP 2008*, 2008.

J.M. Howe and A. King. Efficient groundness analysis in prolog. *Theory and Practice of Logic Programming*, 3, 2003.

ILOG. ILOG cp optimizer. `http://www.ilog.com/products/cpoptimizer/`.

J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Proceedings of the 14th ACM Symposium on the Principles of Programming Languages*, 1987.

C.A. Jefferson, T.W. Kelsey, S.A. Linton, and K.E. Petrie. GAPLex: Generalised static symmetry breaking. In *SymCon'06: The Sixth International Workshop on Symmetry in Constraint Satisfaction Problems*, 2006.

P. Jégou. Decomposition of domains based on the micro-structure of finite constraint-satisfaction problems. In *Proceedings of The Eleventh National Conference on Artificial Intelligence (AAAI-93)*, 1993.

Y.C. Law, J.H.M. Lee, T. Walsh, and J.Y.K. Yip. Breaking symmetry of interchangeable variables and values. In *Principles and Practice of Constraint Programming - CP 2007*, 2007.

E.M. Luks and A. Roy. The complexity of symmetry-breaking formulas. *Annals of Mathematics and Artificial Intelligence*, 41, 2002.

A.K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1), 1977a.

A.K. Mackworth. On reading sketch maps. In *International Joint Conference on Artificial Intelligence*, 1977b.

T. Mancini and M. Cadoli. Detecting and breaking symmetries by reasoning on problem specifications. In *Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, 2005.

K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 2, 1993.

K. Marriott and P.J. Stuckey. Approximating interaction between linear arithmetic constraints. In *Proceedings of the 1994 International Symposium on Logic Programming*, 1994.

K. Marriott and P.J. Stuckey. *Programming With Constraints: An Introduction*. MIT Press, 1998.

I. McDonald and B. M. Smith. Partial symmetry breaking. In *Principles and Practice of Constraint Programming - CP 2002*, 2002.

B.D. McKay. Practical graph isomorphism. *Congressus Numerantium*, 30, 1981.

C. Mears, M. Garcia de la Banda, B. Demoen, and M. Wallace. Lightweight dynamic symmetry breaking. In *SymCon'08: The Eighth International Workshop on Symmetry in Constraint Satisfaction Problems*, 2008a.

C. Mears, M. Garcia de la Banda, M. Wallace, and B. Demoen. A novel approach for detecting symmetries in CSP models. In *Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2008b.

C. Mears, M. Garcia de la Banda, M. Wallace, and B. Demoen. Symmetry breaking and branching constraints. In *SymCon'08: The Eighth International Workshop on Symmetry in Constraint Satisfaction Problems*, 2008c.

C. Mears, M. Garcia de la Banda, and M. Wallace. On implementing symmetry detection. *Constraints*, 14, 2009.

P. Meseguer and C. Torras. Exploiting symmetries within constraint satisfaction search. *Artificial Intelligence*, 129(1-2), 2001.

K. Muthukumar and M.V. Hermenegildo. Combined determination of sharing and freeness of program variables through abstract interpretation. In *Proceedings of the Eighth International Conference on Logic Programming*, 1991.

K. Petrie. Further analysis of SBDS against SBDD. In *SymCon'03: The Third International Workshop on Symmetry in Constraint Satisfaction Problems*, 2003.

K.E. Petrie and B.M. Smith. Symmetry breaking in graceful graphs. In *Principles and Practice of Constraint Programming - CP 2003*, 2003.

G. Puebla, E. Albert, and M. Hermenegildo. A generic framework for the analysis and specialization of logic programs. In *Proceedings of the 15th Workshop on Logic-based methods in Programming Environments*, 2005.

J.-F. Puget. On the satisfiability of symmetrical constrained satisfaction problems. In *Methodologies for Intelligent Systems, 7th International Symposium, ISMIS '93*, 1993.

J.-F. Puget. Symmetry breaking revisited. In *Principles and Practice of Constraint Programming - CP 2002*, 2002.

J.-F. Puget. Automatic detection of variable and value symmetries. In *Principles and Practice of Constraint Programming - CP 2005*, 2005a.

J.-F. Puget. Breaking symmetries in all different problems. In *International Joint Conference on Artificial Intelligence*, 2005b.

J.-F. Puget. Breaking all value symmetries in surjection problems. In *Principles and Practice of Constraint Programming - CP 2005*, 2005c.

J.-F. Puget and B. Smith. Improved models for graceful graphs. In *CP 2006 Workshop on Constraint Modelling and Reformulation*, 2006.

A. Ramani and I.L. Markov. Automatically exploiting symmetries in constraint programming. In *CSCLP 2004: Joint Annual Workshop of ERCIM/CoLogNet on Constraint Solving and Constraint Logic Programming*, 2004.

J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of The Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1994.

C.M. Roney-Dougal, I.P. Gent, T. Kelsey, and S. Linton. Tractable symmetry breaking using restricted search trees. In *ECAI 2004: 16th European Conference on Artificial Intelligence*, 2004.

F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming.* Elsevier, 2006.

P. Roy and F. Pachet. Using symmetry of global constraints to speed up the resolution of constraint satisfaction problems. In *ECAI98 Workshop on Non-binary Constraints*, 1998.

M. Sellmann and P. Van Hentenryck. Structural symmetry breaking. In *International Joint Conference on Artificial Intelligence*, 2005.

B.M. Smith. Reducing symmetry in a combinatorial design problem. Technical report, School of Computer Studies, University of Leeds, Jan 2001.

B.M. Smith. Caching search states in permutation problems. *Principles and Practice of Constraint Programming - CP 2005*, 2005.

B.M. Smith. Constraint programming models for graceful graphs. In *Principles and Practice of Constraint Programming - CP 2006*, 2006.

G. Smolka. The Oz programming model. In *Computer Science Today*. 1995.

G. Szilagyi, T. Gyimothy, and J. Maluszynski. Static and dynamic slicing of constraint logic programs. *Automated Software Engineering*, 9, 2002.

P. Van Hentenryck. *The OPL optimization programming language*. MIT Press, 1999. ISBN 0-262-72030-2.

P. Van Hentenryck and L. Michel. *Constraint-Based Local Search*. MIT Press, 2005.

P. Van Hentenryck and L. Michel. The steel mill slab design problem revisited. In *Fifth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2008.

P. Van Hentenryck, P. Flener, J. Pearson, and M. Agren. Compositional derivation of symmetries for constraint satisfaction. In *Proceedings of the International Symposium on Abstraction, Reformulation and Approximation (SARA 2005)*, 2005.

D. Voets and D. De Schreye. A new approach to non-termination analysis of logic programs. In *Logic Programming*, 2009.

M.G. Wallace, S. Novello, and J. Schimpf. ECLiPSe : A platform for constraint logic programming. *ICL Systems Journal*, 12(1), 1997.

T. Walsh. SAT v CSP. In *Principles and Practice of Constraint Programming - CP 2000*, 2000.

D. Waltz. Understanding line drawings of scenes with shadows. In *The Psychology of Computer Vision*, 1975.