Representations in Constraint Programming

Christopher Jefferson

This thesis is submitted in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy.

University of York
York
YO10 5DD
UK

Department of Computer Science

June 30th 2007

Abstract

Constraint programming is a powerful and general purpose tool which is used to solve both combinatorial and real-world problems. The process of mapping a real-world problem into a constraint program, called modelling, involves a number of choices and hence it is currently more of an art than a science.

One important choice in modelling is what variables will be used to represent the states of the problem. This choice is often optimised to try to take advantage of the global constraints in the solver being used. This concentration on global constraints has allowed the performance of models to improve as more and better global constraints are designed and implemented. However, this obsession with global constraints has lead to a lack of study of the variables, independent of the particular set of constraints implemented by a given solver.

The results of this thesis provide a strong basis on which to build systems which can automatically choose the appropriate variables to represent the high-level variables in a given problem.

This thesis presents a framework to compare the different sets of variables which represent a given problem, independent of how the constraints are implemented. This provides a number of powerful and useful results and allows a number of useful results to be proved. These include how a common set representation performs as well as the theoretically best possible representation on a large set of common set and multisets constraints. Further, many families of constraints on sets and multisets which include the cardinality constraint are shown to have no tractable implementation. This provides a limit on how well representations can perform.

This framework for describing representations is also used to show how a close examination of the variables can allow more efficient implementations and better understanding of common concepts in constraint programming, including connecting multiple models of the same problem and breaking symmetry.

Acknowledgements

There are many people who have helped me while working through this thesis. Obviously I would first like to thank Alan Frisch, who has tolerated my many diversions & wanderings and provided the topics this thesis is about. I want to thank Ian Miguel for being a constant source of useful suggestions and paper references from the next desk, and along with his wife Angela providing nights filled with horror films and Neo-Geo gaming. I would also like to thank Ian Gent for lending his ideas and name to the "Gent Representation", Pete Jeavons for both useful discussions and tolerating my thesis overflowing far longer than it was ever intended to, my CP drinking group for the "eye rule" and the York Anime Society for giving me something to do on Sundays. I also want to thank my family, in particular my parents and brothers, for their support through my very lengthy education.

I am in debt to both of my examiners, Christian Bessiere and Colin Runciman, for providing a large number of helpful corrections, and tolerating a submitted thesis which was less polished than I would have liked. My apologises for requiring you to slog through quite so many minor mistakes, and obviously any remaining mistakes are my own!

Finally, I have no doubt that without Karen's everlasting fountain of love, help and pressure this thesis would have been much worse, much later, and most likely never finished at all. I love you with all my heart.

Declarations

Parts of this thesis have appeared in the following publications which have been subject to peer review:

1. Section 8.4 is based on and extends:
   Alan M. Frisch, Christopher Jefferson and Ian Miguel. *Symmetry-breaking as a Prelude to Implied Constraints: A Constraint Modelling Pattern*, Proceedings of the 16th European Conference on Artificial Intelligence, 2004.

2. Parts of Sections 8.1 and 8.2 are based on:
   Dave Cohen, Peter Jeavons, Chris Jefferson, Karen E. Petrie and Barbara M. Smith. *Symmetry Definitions for Constraint Programming*, Journal of Constraints, 11:115-137, 2006.

3. Throughout this thesis is work based on and extending:
   Christopher Jefferson and Alan M. Frisch. *Representations of Sets and Multisets in Constraint Programming*, Proceedings of the Fourth International Workshop on Modelling and Reformulating Constraint Satisfaction Problems, LNCS 3709, Springer, 2005.

Contents

List of Definitions

Chapter 1

Introduction

"Science is what we understand well enough to explain to a computer. Art is everything else we do."

Donald Knuth, Foreword to $A = B$.

At its core, Computer Science is the study of algorithms which enable computers to perform tasks. Artificial Intelligence is the study of "intelligent" algorithms, which can be used to solve a range of different problems with varying levels of effectiveness. One major breakthrough in this area came from Stephen Cook [14], who showed the existence of problems which can be used to express a large range of other problems and solve them. These problems, found by Cook, therefore provide the possibility of an effective general framework to solve a huge range of further problems.

There is a large number of these problems. Some of them, including Satisfiability (SAT), Integer Linear Programming (ILP) and Constraint Programming (CP), have been practically investigated and used as a universal framework to solve both real-world and combinatorial problems.

All three of these languages require a problem to be specified as a list of variables, each with a domain and a list of constraints on the variables. Both SAT and ILP provide a very small language of allowed constraints, requiring the use of a library of special tricks to map more complex constraints and objects. CP overcomes this problem by not fixing the list of allowed constraints, but instead allowing them to vary from problem to problem. This means CP provides a much more expressive language than SAT or ILP. CP solvers come with a large selection of efficient implementations of many constraints and often provide the option of implementing new constraints.

Unfortunately while mapping a problem to some CP models is simple, these naive models will usually perform extremely poorly. There are a number of choices which must be made when constructing a good CP model of a given problem. Firstly, the possible assignments to the problem must be mapped to a number of CP variables. Secondly, the constraints must be mapped to these variables, where possible using those constraints which have an efficient implementation in the solver. It is also possible to implement new problem-specific constraints.

Further, but often necessary, steps in effectively refining a problem to a CP model include adding logically redundant constraints, generating multiple connected models of the same problem and dealing with both the symmetries of the original problem and those introduced during modelling. While experienced practitioners in the field are often able to quickly come up with reasonable choices for each of these tasks, modelling is still very much an art rather than a science.

The flexibility of being able to add new constraints to a solver has provided huge gains in the practical performance of CP solvers, with both commercial [47] and open-source [12, 36, 75] solvers providing a range of global constraints and providing the ability to add new ones. Algorithms to allow the efficient implementation of new constraints are frequently produced, leading to a large catalogue of constraints which can be used to provide more expressive ways of writing problems.

This concentration on implementing ever more powerful constraints has however lead to a gap in the theory of constraint programming. Frequently the quality of a set of variables is considered only in terms of which constraints with efficient implementations from a particular solver can be placed on them.

This thesis takes a different and novel route, building a theory of how well a particular set of variable represents a problem independently of the practical implementation of the constraints. While these two issues cannot be completely separated in practice, studying the variables in isolation allows two interesting and useful observations. Firstly, cases where a given representation has flaws which are impossible to fix by implementing better constraints. Secondly, cases where apparent differences between different representations are only artefacts of the usage of a particular set of global constraints, which can then be easily, efficiently and generally worked around.

One major limitation of ignoring implementation details and concentrating solely on search trees is that it ignores that constraints may be very efficiently implementable on one representation but take exponential time to implement for another representation. This does mean that the results of this thesis will have to be applied with sense. Ignoring this issue does have a number of major advantanges, not least allowing the comparison of representations and families of constraints whose complexity has not been identified, rather than remaining limited to a number of small well-studied representations and constraints where the complexity of implementation is known.

Chapter 2 provides definitions of the mathematical terms and notation which will be used throughout this thesis, as well as a short guide to constraint programming and the common techniques and algorithms used to implement modern backtracking constraint solvers. Finally this chapter provides a brief introduction to group theory and symmetry, which is used only in Chapter 8.

Chapter 3 discusses the sections of the constraint programming literature and some closely related areas which are directly related to this thesis. This will provide an overview of the existing work which has been performed on reformulation and refinement of constraint programs and show where the holes are in this previous work which will be filled in this thesis.

Chapter 4 provides an introduction to the central idea of this thesis, a representation of a single variable by either a lattice or a list of other variables with a mapping back to the domain of the original variable. There are a number of subtle issues which must be dealt with in providing a definition of a representation. Any definition must both encapsulate those representations already in common usage, without allowing structures which would violate conditions which a CP practitioner would expect to hold. This chapter also provides a guide to how representations can be used to help solve a particular CSP and gives a number of examples of representations. In particular, this chapter provides a number of representations of set and multiset variables which will be used throughout the rest of the thesis. While sets and multisets will be the principle example used in this thesis, the theory applies to any kind of representational choice.

Using the definitions built in Chapter 4, Chapter 5 discusses the strongest possible relationship between two different representations of the same variable, that one of them produces a smaller search space for all possible CSPs, which is referred to as one dominating the other. This also leads naturally on to the idea of two representations being equivalent. This chapter discusses why different equivalent representations may appear to differ, and how these changes can be efficiently dealt with.

While dominance provides a very strong method of comparing representations, in many cases it is too coarse. Chapter 6 shows how representations can be compared on all CSPs built using a particular language of constraints. This provides a much finer comparison. The main result of this chapter is to give a number of sufficient conditions for one representation to be as good as the perfect, or complete, representation on a given language of constraints. This means that on CSPs built from only those constraints, such representations generate the smallest possible search space.

Chapter 7 differs from the other chapters of the thesis in providing no theoretical results. This chapter contains a number of experiments, comparing a common set representation with randomly generated ones on both a common structured CP problem known as the Balanced Incomplete Block Design and randomly generated problems. This chapter does not aim to provide a comprehensive comparison of representations, but aims to illustrate a number of interesting features of representations and also channelling between different representations.

Symmetry is an important feature of solving many constraint problems. Chapter 8 considers how symmetry breaking interacts with representational choice. This results in a better understanding of how choosing an appropriate method of breaking symmetry can result in smaller searches and and also provides new symmetry breaking methods can proactively break symmetry, using some of the theory of representations presented earlier in the thesis. This can provide smaller search spaces than existing symmetry breaking methods which mostly only act reactively, avoiding parts of the search space symmetric to those explored before.

Chapter 2

Background

This chapter introduces a number of concepts used throughout this thesis. Section 2.1 introduces the general mathematical terms and definitions and the rest of the chapter gives a general introduction to the mathematics and algorithms of constraint programming.

2.1 Mathematical Introduction

This section introduces some necessary basic mathematical concepts which are used throughout the thesis. Many of these definitions will already be familiar, however there sometimes slight differences in how these terms are defined and so they are given here to avoid any confusion. Definition 2.1 defines some basic mathematical terms.

Definition 2.1. $X \iff Y$ denotes "if and only if", and is true only when either both X and Y are true, or both are false.

$X \equiv Y$ is true if and only if either $X = Y$ or both X and Y are undefined.

Two of the most used concepts in this thesis are sets and multisets. While these are common terms, the exact definitions are given in Definition 2.2. The definitions of a number of common operators used on sets and multisets are given in Definition 2.3.

Definition 2.2. A **multiset**, denoted as a list of elements $\{x_1, x_2, \ldots\}_m$, is an unordered collection which permits duplicates. The more common notation, {{}}, for multisets is not used to avoid confusion with sets of sets. A **set** can be considered as a multiset in which no value can occur more than once. Sets can be denoted by a list of distinct elements between $\{\}$.

Definition 2.3. Given sets or multisets S and T, the following operations are defined:

Element: $occ(i, S)$ denotes the number of occurrences of i in the set or multiset S. Of course this value will always be zero or one for a set. $i \in S$ denotes that there is at least one occurrence of i in S.
$occ(1, \{1, 2, 1\}_m) = 2,\ occ(1, \{2, 3\}) = 0$

Subset: $S \subseteq T$ is true if the number of occurrences of all values in S is less than the number of occurrences in T
$\{1, 1, 2\}_m \subseteq \{1, 1, 2, 2\}_m,\ \{1, 1, 2\}_m \not\subseteq \{1, 2, 2\}_m$

Intersection: $S \cap T$ is the (multi)set which for each i in either S or T contains the minimum of $occ(i, S)$ and $occ(i, T)$.
$\{1, 1, 2, 2\}_m \cap \{1, 1, 1, 2, 3\}_m = \{1, 1, 2\}_m$

Union: $S \cup T$ is the (multi)set which for each i in either S or T contains the maximum of $occ(i, S)$ and $occ(i, T)$.
$\{1, 1, 2, 2\}_m \cap \{1, 1, 1, 2, 3\}_m = \{1, 1, 1, 2, 2, 3\}_m$

Other set and multiset operation will be defined where they are used.

Definition 2.4 defines some methods for referring to sets of numbers. As mentioned in the definition, rational, real and complex numbers are not used in this thesis unless explicitly stated otherwise. This is not required by the theory, but makes its presentation much simpler.

Definition 2.4. $\mathbb{Z}$ denotes the set of integers. $\mathbb{N}$ denotes the set of integers greater than or equal to 1 and $\mathbb{N}_0$ denotes the set of integers greater than or equal to 0. The notation $[a..b]$ for integers a and b where $a \leq b$ refers to the set of integers $\{i|a \leq i \leq b\}$. It can be assumed all examples in this thesis will only use integers rather than the rational numbers ($\mathbb{Q}$) or real numbers ($\mathbb{R}$) unless explicitly stated otherwise.

Relations are very important in this thesis, used in definitions of both constraints and representations. Definition 2.5 gives the formal definition of a relation which will be used in this thesis and a number of special types of relations which will arise.

Definition 2.5. Given a set S, a **relation** R on S is a set of ordered pairs (i, j) where both i and j are from S. There are a number of special conditions a relation can satisfy:

Symmetric: $(i, j) \in R \iff (j, i) \in R$.

Anti-symmetric: $(i, j) \in R \wedge (j, i) \in R \implies i = j$.

Transitive: $(i, j) \in R \wedge (j, k) \in R \implies (i, j) \in R$.

Reflexive: $\forall i.\ (i, i) \in R$.

Partial Ordering: Anti-symmetric, transitive and reflexive.

Total Ordering: An ordering where at least one of $(i, j) \in R$ and $(j, i) \in R$ is true for all pairs of values i, j in S.

There are many different names for ordered lists. In this thesis these will always be called arrays (Definition 2.6), rather than vector or list.

Definition 2.6. An array is denoted as $\overline{w}$. The elements are listed explicitly as $\langle a_1, a_2, \ldots, a_n \rangle$. The length of $\overline{w}$ is denoted $|\overline{w}|$. The i^{th} element is denoted by $w[i]$. $\overline{w[a \ldots b]}$ represents the array $\langle w[a], w[a + 1], \ldots, w[b] \rangle$.

While functions are a basic and commonly used concept, there are a number of slightly different definitions in common usage. Definition 2.7 gives the exact definitions used for various types of functions in this thesis.

Definition 2.7. A **function** from a set S to a set T is denoted $f : S \to T$. A **partial function** is a function which is only defined for some values in S. A **total function** is defined for all elements of S (functions will be total unless stated otherwise). The **domain** of a function is those values of S for which it is defined, the **range** is the set of t in T such that $f(s) = t$ for some s. Three frequently used types of functions are:

surjective: For all t in T, there is some s such that $f(s) = t$.

injective: For any two distinct elements s_1 and s_2 of S, $f(s_1)$ and $f(s_2)$ are different.

bijective: A function which is both injective and surjective.

Given a bijection $g : S \to T$, the **inverse** of g, denoted g^{-1}, is the function from T to S defined by $g^{-1}(t) = s \iff g(s) = t$.

Given $f : S \to T$ and $g : T \to U$, the **composition** of f and g, often denoted gf, is a function mapping S to U defined as $gf(x) = g(f(x))$.

Often functions are not defined simply between sets, but between structured objects and in this case it often only makes sense to consider functions which preserve structure in some way. Homomorphisms are one such class of function.

Definition 2.8. Given two sets S and T with a relation R_S on S and R_T on T, a **homomorphism** is a function f from S to T such that if the pair (s_1, s_2) is in the relation R_S, then the pair $f(s_1), f(s_2)$ is in the relation R_T.

One fundamental mathematical concept, which will be used throughout this thesis, is sets and their generalisation multisets, both of which are defined in Definition 2.2. In general sets will be considered as multisets which are restricted to only allow at most one of each value. While this notation is unusual for sets, it allows many parts of the thesis to easily discuss both sets and multisets simultaneously. Example 2.1 gives some examples of sets and multisets.

Example 2.1. *Both* $\{1,2\}, \{2,1\}$ *are the same set which contain exactly two elements.* $\{1,2\}_m$ *represents the multiset containing the same elements.* $\{1,2,2\}$ *is not a valid set.* $\{1,2,2\}_m$ *is a multiset containing the one occurrence of* 1 *and two occurrences of* 2. $occ(1, \{1,2\}) = 1$ *and* $occ(0, \{1,2\}) = 0$.

There is a large number of ways of building new sets and multisets from existing ones. A number of these will be used in thesis, which are given in Definition 2.9. Example 2.2 shows these methods being used on some small sets and multisets.

Definition 2.9. $\{f(x)|p(x)\}$ denotes the set S defined as $f(x) \in S \iff p(x)$. $\mathbb{P}(S)$ is the powerset of S, defined as the set $\{s|s \subseteq S\}$.

Example 2.2. *Both* $\{x^2|0 \leq x \leq 3\}$ *and* $\{x^2|-3 \leq x \leq 3\}$ *are equal to the same set,* $\{0,1,4,9\}$[1]. *Power-sets exist for both sets and multisets. For example:*

- $\mathbb{P}(\{0,1\}) = \{\{\}, \{0\}, \{1\}, \{0,1\}\}$.
- $\mathbb{P}(\{0,1,1\}_m) = \{\{\}_m, \{0\}_m, \{1\}_m, \{0,1\}_m, \{1,1\}_m, \{0,1,1\}_m\}$.

[1]All examples in this thesis will only consider integer values, as already stated

There are many restricted families of set and multiset domains, for example where all the assignments must have the same size. Some of these families have better representations which can make use of the extra information. The families which will be used in this thesis are described in Definition 2.10.

Definition 2.10. Given a set S and $c \in \mathbb{N}$, the following list gives a number of special set and multiset domains and their description in English. These are outlined as they will be used repeatedly during this thesis.

- $\mathbb{P}(S)$
  All subsets of a set S.
- $\{s \mid s \subseteq S, |s| = c\}$
  All subsets of a set S of size c.
- $\{s \mid s \subseteq S, |s| \leq c\}$
  All subsets of a set S of size at most c.
- $\{s \mid s \subseteq_m S, \forall i.\ occ(i, s) \leq c\}$
  All multisets containing at most c occurrences of each element of a set S.
- $\{s \mid s \subseteq_m S, |s| = c\}$
  All multisets drawn from a set S of size c.
- $\{s \mid s \subseteq_m S, |s| = c\}$
  All multisets drawn from a set S of size at most c.

2.2 Constraint Satisfaction Problems

Constraint satisfaction problems, often abbreviated to CSP, are a commonly used, natural and expressive manner of specifying both combinatorial and real-world problems. A CSP is defined in 3 parts, an array of *variables*, each with an associated *domain*, and a set of *constraints* which the variables must satisfy in a solution.

The first usage of CSPs is generally considered to be by Ugo Montanari [60] as a method of describing and solving problems related to line drawings. Since then a number of practical and powerful algorithms have been invented and used to solve a large range of problems. The fundamental definition of a finite constraint satisfaction problem remains similar, it is given in Definition 2.11 and a simple example is given in Example 2.3. Throughout this thesis only finite domain CSPs will be considered, so the requirement of finiteness will not be mentioned again.

Definition 2.11. A **Constraint Satisfaction Problem** (CSP) P is a triple $\langle \overline{V}, D, C \rangle$ where:

1. $\overline{V}$ is an array of **variables**.
2. D is a function which maps each variable to a set, called the **domain** of the variable. The domain of a variable v is also denoted $dom(v)$.
3. C is a set of **constraints**.

A **sub-domain** of a variable v is a subset of $D(v)$. A **literal** is a pair $\langle v, d \rangle$ where v is a variable and d is a value from its domain. An **assignment** is a sub-domain of size 1. An assignment of an array of variables consists of an assignment to each element of the array, similarly an assignment to a CSP consists of an assignment for all the variables in the CSP. It is hoped this overloading of the term assignment to apply to single variables, array of variables and whole CSPs aids rather than confuses the reader.

Each constraint c is defined over an array of variables, call the **scope** of c, denoted $scope(c)$. The arity of a constraint is defined as $|scope(c)|$. Constraints of arity 1 are called **unary**, those of arity 2 are called **binary**. Any assignment to the variables in $scope(c)$ will either satisfy or fail to satisfy the constraint. An assignment to an array of variables which includes $scope(c)$ satisfies c if and only if the values that assignment takes on $scope(c)$ satisfies c. An assignment to P which satisfies all the constraints is a **solution**.

Example 2.3. *Consider the CSP $P = \langle V, D, C \rangle$, defined as:*

$$V = \{X, Y, Z\}.$$

$$D = X \mapsto \{1,2,3\}, Y \mapsto \{1,2,3\}, Z \mapsto \{1,2,3\}.$$

$$C = \{\mathbf{X} > \mathbf{Y}, \mathbf{X} + \mathbf{Y} > \mathbf{3}, \mathbf{X} * \mathbf{Y} * \mathbf{Z} \geq \mathbf{6}\}.$$

The partial assignment $\langle X, Y \rangle = \langle 2, 1 \rangle$ satisfies the constraint $\mathbf{X} > \mathbf{Y}$ but does satisfy $\mathbf{X} + \mathbf{Y} > \mathbf{3}$. As this partial assignment does not give an assignment to Z, it is not defined if this partial assignment satisfies $\mathbf{X} * \mathbf{Y} * \mathbf{Z} \geq \mathbf{6}$. The assignment $\langle X, Y, Z \rangle = \langle 2, 1, 3 \rangle$ satisfies all the constraints and is therefore a solution.

Definition 2.11 differs from the normal definition of a CSP by not concretely defining what a constraint is. Usually a constraint is defined as a set of allowed assignments as in Definition 2.12. The problem with this definition is that modern constraint solvers express many constraints in an intensional language, for example arithmetic constraints such as $\mathbf{A} = \mathbf{B}$, or $\mathbf{A} + \mathbf{B} + \mathbf{C} = \mathbf{3}$ or more complex statements like "there are at least 4 occurrences of X in the array $\overline{L}$". In most of this thesis this distinction will not be of importance, where it matters it will be discussed.

Definition 2.12. An **extensional constraint** $c \in C$ in a CSP $P = \langle V, D, C\rangle$ is a pair $\langle \overline{W}, T\rangle$ where each element of $\overline{W}$ is an element of $\overline{V}$ and T is a set of assignments to $\overline{W}$. An assignment $\overline{a}$ of $\overline{W}$ is accepted by c if and only if $\overline{a} \in T$.

A *CSP solver* takes a CSP instance and searches for assignments to each variable which together satisfy all of the constraints. This basic framework can be extended in a number of ways, one of the most common is by allowing an *optimisation function* which maps assignments to the integers, and requiring the solver to find the solution which has the highest (or lowest) value under this optimisation function. There are a range of both commercial [47] and open-source [12, 36, 75] general-purpose constraint solvers which can be used to solve CSPs.

2.2.1 Complexity of CSPs

One reason why CSPs are so important is that they are an example of an *NP-complete* language which is expressive and has a number of highly efficient solvers. NP-complete problems are important theoretically as they provide a method of solving a large range of problems. Definition 2.13 gives an overview of the basic definitions involved in NP and NP-completeness. A more complete discussion of complexity and NP-completeness can be found in Garey and Johnson [32].

Definition 2.13. Given an infinite class of problems C, where the input size of each instance I in C is denoted $|I|$:

- C is **polynomial time solvable** if there exists a polynomial p such that any instance I can be solved in time $p(|I|)$.
- C is NP-easy if there exists a polynomial p such that possible solutions to an instance I ca be checked in time $p(|I|)$.

- C is NP-hard if given any NP-easy problem D, there is a function $f : D \rightarrow C$ such that $f(d)$ is solvable if and only if d is and there exists a polynomial p such that $|f(d)| \leq p(|d|)$.
- C is NP-complete if it is both NP-easy and NP-hard.

If a problem is NP-complete, any NP-easy problem can be mapped to it with only a polynomial increase in the required space. As an NP-easy problem requires only it is possible to check in polynomial time if a possible solution is correct, this includes a huge range of problems. At the present moment, it is not known if there is an algorithm to solve NP-complete problems in polynomial time, and it appears unlikely such a proof will be provided soon. Therefore the best known algorithms for solving NP-complete problems are all exponential in the worst case. However, there can be a large variance between the performance of the known algorithms in practice.

One of the first problems to be proved NP-complete, by Cook's theorem [15], and a common language for mapping other problems into is satisfiability, commonly known as SAT. Definition 2.14 provides a definition of the SAT problem, and a few variants.

Definition 2.14. An instance of SAT is a pair $\langle V, C \rangle$, where V is a set of Boolean variables, and C is a set of clauses. A clause is a set of literals, where a literal is either a variable or it's negation. Given an assignment to V, a clause is satisfied if at least one literal in it is true. A solution is an assignment to V which satisfies all the clauses.

k SAT is a variant of SAT where every clause must be of size k. 1-in-k SAT requires each clause be of size k, and exactly one literal is true in each clause. Both k SAT and 1-in-k SAT are NP-complete, as shown in [32].

Example 2.4. *The SAT problem $(A \wedge B) \vee (!A \wedge C)$ is true when A and C are both true, for any value of B. Considered as a 1-in-2 SAT problem, B would have to be false to ensure in the first clause only one term was true.*

There are a number of SAT solvers available [20, 61] which can solve problems with hundreds of thousands of variables, which are used to solve problems in a number of fields.

CSPs can easily be seen to be NP-easy as long as each constraint is NP-easy, as checking a solution to an CSP just involves checking each constraint in turn.

A 3-SAT problem is obviously a CSP, so CSPs are also NP-hard. Therefore they are also NP-complete, while providing a more expressive language than SAT.

While it is possible with only polynomial space increase to map any NP-complete problem to any other, in many cases the transformation is complex and loses much of the structure of the original problem. As CSPs allow any constraint to be used, it is often easier to express problems as a CSP rather than in the much more limited language of SAT. This makes at the language CSPs make them a good candidate for a general framework to easily express problems.

One important place where NP-completeness comes up in constraint programming is propagation algorithms. The propagators of many intensional constraints turn out to be NP-complete and in general these are considered too expensive, so a less powerful form of propagation must be used. This is discussed further in Section 2.3.5.

2.3 Solving CSPs

Once a problem has been specified as a CSP the next step is to solve it. Depending on what is required, this may involve proving no solution exists or finding either one solutions, all solutions or the solutions which maximises an objective function. There are a number of different methods for solving CSPs, which vary in the complexity of implementation, speed of solving and which types of problems they are best at solving. As CSPs are NP-complete, in the worst case all known algorithms have at least exponential complexity. This does not stop there being large differences in the relative performance of the various algorithms.

2.3.1 Trivial Methods

It is useful to consider first trivial methods of solving CSPs, as these can then be compared and contrasted with more efficient methods. The most trivial and obvious method of solving a CSP is *generate and test*, which simply enumerates all possible assignments and checks each of them in turn to see if it satisfies all the constraints, and therefore is a solution. This method is trivially correct and as CSPs are NP-complete, has the best known worst-case complexity. However, for many problems it is possible to do much better than trivial algorithms such as this.

2.3.2 Stochastic Search

One family of methods for solving CSPs are incomplete methods [66]. These algorithms typically start with a random assignment to the variables of the CSP, and then use varying methods to move through the space of assignments in the search for a solution. Many variations of stochastic search exist, which use different methods of moving through the space of assignments and also use various techniques to restart search in a different part of the search space.

The biggest flaw of stochastic search is that it is impossible to ever show that all solutions to a problem have been found, or to show a problem has no solutions. More importantly to this thesis while how variables are represented appears to be as important in stochastic search as it is in complete search; it is non-obvious what connections, if any, there are between the theory of representations in stochastic and backtrack search. For example in complete search symmetry breaking has proved to almost always be beneficial but in stochastic search Prestwich and Roll [63] showed that symmetry breaking frequently leads to poorer performance and purposefully adding extra symmetry to a problem can improve performance. This thesis considers only search strategies which build on backtracking search and stochastic search will not be considered or discussed further.

2.3.3 Backtracking Search

Chronological backtracking (BT) [8] is the foundation of almost all modern CSP solvers and algorithms. BT is a tree-search algorithm which tries each possible assignment to each variable in turn, and backtracks if the partial assignment at any node violates any constraint. An example of a BT search for the CSP given in Example 2.3 on page 19 is given in Figure 2.1. This diagram shows how search progresses by choosing a variable and branching on each value in its domain. The large circular nodes represent failures and the square node finding a solution.

This figure shows some interesting features of BT searches. Much less search is performed for $X = 1$ than would be done for generate and test, because as soon as Y is assigned, the constraint $X > Y$ is violated.

This basic idea of BT can be improved upon in two ways. Firstly in Example 2.3 the constraint $\mathbf{X} > \mathbf{Y}$ clearly implies that in no solution can $X = 1$ or $Y = 3$. This can be used to reduce the domains of X and Y before search and similar reasoning could be applied during search if the domains of X or Y changed. This

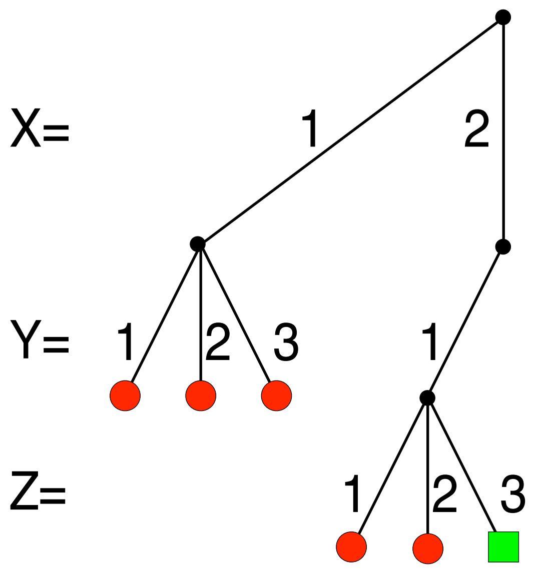

Figure 2.1: A BT search tree of Example 2.3.

is the basis of *propagation algorithms*, a group of algorithms which have led to many magnitudes of improvement in the performance of CSP solvers. Secondly the order in which the BT algorithm choses which variables and values to branch on can have a massive effect in the size of the search tree. In the extreme case, if the variable and values are chosen in exactly the correct order the search can move directly to a solution.

2.3.4 Branching

Chronological backtracking branches at each node by taking a variable and generating a new search node for every assignment (Definition 2.11 on page 19) to that variable. There are more complex methods of branching which can greatly reduce the size of search. Definition 2.15 gives a general definition of static and dynamic search methods. Static branching methods are simpler, as they are fixed before search and are dependent only on the current depth of search. Note that the definitions of static and dynamic branching algorithms given here are not sufficient to ensure that search will actually terminate. This is because the results of going down a branch requires discussing how the constraints are propagated, and so must wait until later.

Definition 2.15. A **branching algorithm** defines how during search in a CSP the solver generates the child nodes of the current node, by adding a new constraint to each child node.

A **static branching algorithm** on a set of variables V is an array L of arrays of constraints where each element C of L must satisfy the condition that any assignment to the variables of the CSP must satisfy at least one constraint in C.

A **dynamic branching algorithm** on a set of variables V is an algorithm which for any node of search uses the state of search at that node to generate an array of constraints C such that every assignment to the variables of the CSP is satisfied by at least one element of C.

There are a number of common branching methods in common usage. The main one considered in this thesis is given Definition 2.16.

Definition 2.16. Given an array of variables $\overline{V}$, and an ordering of the domain of each element of V, the **N-way static branching** takes the first variable which has not yet been branched on in $\overline{V}$ and generates a branch assigning that variable each value in its domain in turn.

There is a large body of research into heuristics which have shown they are of vital importance to efficiently solving CSPs. Dynamic search heuristics are often found to lead to smaller search trees [58] and Hwang and Mitchell [46] showed that 2-way branching can lead to exponentially smaller searches than an n-way branching heuristic in some CSPs, although this result only holds if adding auxiliary variables to the CSP is forbidden.

Unfortunately, the huge flexibility of general dynamic search heuristics and the unpredictability they bring to how the search tree will be generated makes it very difficult to compare different representations when dynamic heuristics are being used. In particular, Theorem 5.1 on page 75 shows the simple result that given any two representations there exist dynamic branching algorithms which will lead to the first outperforming the second and the second outperforming the first in finding a first solution.

Often the extreme flexibility of allowing any constraint to be used in branching is not required or at least not used. Definition 2.17 gives the much more commonly used and much simpler branching methods used in many CSP solvers, where the only constraints used in branching are $\mathbf{X} = \mathbf{a}$ and $\mathbf{X} \neq \mathbf{a}$, for a variable X and constant a. This kind of branching was used in the example search in Diagram 2.1 and Definition 2.16.

Definition 2.17. Given a CSP $P = \langle V, D, C \rangle$, a **variable ordering** is a permutation of the elements of V, a **value ordering** for a variable $v \in V$ is a permutation of the domain of $D(v)$.

Given a variable ordering R and a value ordering L_v for each variable, the **N-way branching** static branching algorithm is defined by an array of constraints C, the same length as R, defined by: $C[i] = \langle R[i] = L_{R[i]}[1], R[i] = L_{R[i]}[2], \ldots \rangle$.

2.3.5 Propagation Algorithms

One of the most successful methods for improving chronological backtracking has been propagation algorithms. Propagation is an example of a *forward step* algorithm, which looks at the branching decisions made so far and attempts to make deductions which either reduce the size of the search tree or prove no solution can exist below the current node of search.

Some forward steps, such as path consistency [59] and k-consistency [25] treat the constraints as sets of allowed assignments which can have elements removed if they can be proved to never be in a solution. While very powerful, these methods are unusable if constraints are not represented as a set of assignments and very few modern CSP solvers do this. Therefore such methods will not be considered in this thesis.

The other standard method of implementing propagation algorithms is to store a sub-domain for each variable which lists currently allowed assignments. A propagator can remove a value from the sub-domain of a variable when it can be proved it occurs in no solution. This definition varies slightly from the one used in many papers, where the domains are reduced and each node represents a different CSP. This makes a number of theoretical issues much simpler, in particular it fixes the representation for the whole problem, rather than having it change at each node.

Most propagation algorithms used in modern CSP solvers and all encountered in this thesis work in the same basic way. The algorithms operate on a sub-domain of each variable in the CSP and look for domain values which cannot occur in any solution and can therefore be removed. Throughout this thesis unless stated otherwise propagation algorithms operate on only a single constraint at a time, so the only way the propagators of different constraints can pass information is by reducing the current sub-domain of any variable. These propagation algorithms are then performed repeatedly until none of them can remove any more values from the current sub-domain of any variable. Example 2.5 provides some simple examples of propagation algorithms. The second case here shows how it can be necessary to run propagators multiple times until a fixed point is reached. The most important part of this is example is the third, which shows how propagation algorithms which consider each constraint in isolation can miss seemingly simple deductions.

Example 2.5. *The following constraints are each defined on the three variables X, Y and Z which each have domain $\{0, 1, 2\}$.*

1. $\mathbf{X < Y}$
   In any solution, as X must be strictly smaller than Y then X cannot be 2. For a similar reason, Y cannot be 0. As no constraint refers to Z, the domain of Z is unchanged.
   The sub-domains after propagation are $X \in \{0, 1\}, Y \in \{1, 2\}, Z \in \{0, 1, 2\}$

2. $\mathbf{X < Y, Y < Z}$
   Considering only the first constraint, $X \neq 2$ and $Y \neq 0$. Considering only the second, $Y \neq 2$ and $Z \neq 0$. Given $Y \neq 2$, the constraint $\mathbf{X < Y}$ can now be used to deduce that $X = 0$ and $Y = 1$. Using $Y = 1$ and $\mathbf{Y < Z}$, clearly $Z = 2$.
   The sub-domains after propagation are $X \in \{1\}, Y \in \{2\}, Z \in \{2\}$.

3. $\mathbf{X = Y}$, $\mathbf{X \neq Y}$
   Considering each constraint in isolation, no values can be removed from the domain of any variable. Therefore any propagation algorithm which considers constraints in isolation will not deduce these constraints have no solutions.
   The sub-domains after propagation are $X, Y, Z \in \{0, 1, 2\}$.

A series of algorithms have been designed to handle the propagation of CSPs where all the constraints are binary and expressed extensionally, including AC-3 [56] and AC-2001 [7]. These algorithms, in particular AC-2001, can be modified to handle non-binary constraints, but storing non-binary constraints as lists of tuples rapidly becomes impractical, as such an algorithm must have d^n worst-case complexity for a constraint of arity n and domain size d, as this is the number of possible tuples.

Most constraint solvers in practice use a general framework similar to AC-5 [43], which allows special propagators to be written for each constraint, using the kind of reasoning given in 2.5. There are various different methods of implementing such frameworks which attempt to improve performance by checking constraints in different orders or trying ensure the same constraint is not checked multiple times. For the purposes of this thesis these optimisations can be ignored, as they do not affect the theoretical complexity of the algorithms.

While in general propagating a constraint to the maximum possible extent must be exponential, for many constraints it is possible to write special algorithms which can perform propagation in polynomial time, in a similar fashion to Example 2.5. Almost all modern constraint solvers, including ILOG Solver [47], Eclipse [12] and Minion [36] provide a large set of such propagators.

These propagators may perform varying levels of propagation, depending on the level which can be efficiently performed. Often there are multiple propagators implemented for the same constraint, achieving different levels of propagation. Definition 2.18 gives a general definition of a propagator, based on the definition given by Apt in [3]. The only difference between Definition 2.18 and Apt's definition is the fourth requirement. This simplifies using propagators during search as without this identifying solutions requires an extra step, where each assignment found by search is checked against each constraint to see if it satisfies them. An example propagation is given in Example 2.6.

Definition 2.18. Given a constraint c whose scope is the array of CSP variables $\overline{V}$, a **propagator** P of c is a function from the set $S = \{\langle s_1, s_2, \ldots, s_{|\overline{V}|}\rangle | s_i \in dom(V[i])\}$ to itself which satisfies the following set of conditions:

1. Monotonic:
   $$\begin{aligned} \forall\{\overline{s_1}, \overline{s_2}\} \subseteq S.\quad & (\forall i \in \{1, \ldots, |\overline{V}|\}.\ s_1[i] \subseteq s_2[i]) \implies \\ & (\forall i \in \{1, \ldots, |\overline{V}|\}.\ P(s_1)[i] \subseteq P(s_2)[i]) \end{aligned}$$
2. Non-increasing:
   $\forall \overline{s} \in S.\ \forall i \in \{1, \ldots, |\overline{V}|\}.\ P(\overline{s})[i] \subseteq s[i]$.
3. Preserves solutions:
   For all $\overline{s}$ in S and all assignments $\overline{a}$ to $\overline{s}$, if $\overline{a}$ satisfies c, then $\overline{a}$ is also an assignment to $P(\overline{s})$.
4. Identifies Assignments:
   For all $\overline{s}$ in S which are assignments, then if the assignment is not allowed by c then $P(\overline{v}) = \emptyset$.

The **GAC** propagator for a constraint C is the propagator which returns the smallest possible legal sub-domain for each variable.

Example 2.6. *One propagator for the constraint $X < Y$, is defined as follows:*

Given *a sub-domain D_X for X and D_Y for Y*

$\mathbf{D'_X}$= $\{i | i \in D_X, i < max(D_Y)\}$.

$\mathbf{D'_Y}$= $\{i | i \in D_Y, i > min(D_X)\}$.

Return *the new domain D'_X for X and D'_Y for Y.*

For example, given $D_X = \{1, 3, 5\}$ and $D_Y = \{0, 1, 2, 3\}$, this propagator would return $D'_X = \{1\}$, $D_Y = \{2, 3\}$. This propagator encapsulates the first part of Example 2.5. It can be easily checked that it satsfies all the requirements of being a propagator as set out in 2.18

Many constraints have had specialised propagators written for them, which achieve much better performance than the general propagation algorithm in GAC-Schema. One of the most famous of these is an algorithm by Regin [67], which GAC propagates if an array of variables are all different. The complexity of this algorithm is greater than imposing a clique of not-equals, but achieves more propagation. The all-different constraint occurs so frequently in constraint programming that a number of other implementations have be written, for example Van Beek et. al. [54] present an algorithm for implementing all-different which achieves a weaker form of propagation called *Bounds* propagation. Which level of propagation should be used in which problems is something which at present is decided only by trial and error.

2.4 Symmetry Breaking

Many CSPs have symmetry and dealing with it effectively can produce substantial improvements in solution time. The usual definition of symmetry used in CP is an automorphism of the literals of the CSP that maps solutions to solutions. When a CSP has symmetry, symmetric solutions will be found. In many ways more importantly, the solver will spend time searching parts of the search tree that are symmetric. Therefore using a method which breaks some or all of the symmetry of a CSP should reduce the size of the search tree and if the overhead of symmetry breaking is small in comparison to the gains in size of the search tree reduce the time taken to find a solution, or prove no solution exists.

2.4.1 Group Theory

Any discussion of symmetry must involve a discussion of groups, which are the most natural method of representing collections of symmetries. Group theory is an important and long-standing area of mathematical research. Groups are useful in CP as they can be used to express and work with symmetries. Many powerful algorithms for solving group-theoretic questions are known, and have been efficiently implemented in systems such as GAP [31] and MAGMA [9]. Some group theory will be used in the review of previous symmetry breaking methods in Chapter 3, and then extensively in Chapter 8 to extend existing symmetry breaking methods. One special type of group is the permutation group, given in Definition 2.19. From here, the word "group" will be used only to refer to permutation groups. This is not actually a limitation, as all groups that will occur will naturally be permutation groups over some set of elements and in fact all groups can be considered as permutation groups, and must be given in this way to GAP or MAGMA.

Definition 2.19. A permutation group G over a set S is a set of automorphisms on S which satisfy the following axioms:

Identity: $\exists e \in G.\ \forall s \in S.\ e(s) = s.$

Inverse: $\forall g \in G.\ g^{-1} \in G$

Composition: $\forall g, h \in G.\ gh \in G.$

A sub-group of G is a subset of G which also satisfies the group axioms. The order of a G, denoted $|G|$, is the number of elements of G.

The image of an element $s \in S$ under a group element $g \in G$ is denoted s^g.

There is a number of number of ways of manipulating groups. Many of the ones which will be used in this thesis are given in Definition 2.20

Definition 2.20. Given a permutation group G over a set S, the following properties can be defined:

1. Stabiliser : Given $s \in S$, the subset of G defined by $\{g | g \in G, s^g = s\}$ is the stabiliser of s. It forms a subgroup of G.
2. Orbit : Given $s \in S$, the subset of S defined by $\{s^g | g \in G\}$, also denoted as s^G, is the orbit of s in G

Example 2.7. *Given a matrix with m rows and n columns, the row symmetries of the matrix are those permutations which reorder the rows while leaving the elements of each column in the same position. The column symmetries are defined similarly. Row and column symmetry is the combination of all elements of these two groups. Note that this group does not contain all symmetries of the matrix, for example the elements of each row must still appear in the same row, although possibly reordered.*

2.4.2 Symmetry Definitions

Definition 2.21. Given a set D and some predicate P defined over the elements of D, a symmetry is defined as a bijective function $f : D \to D$ such that $\forall x \in D, P(x) \iff P(f(x))$.

As f is bijective it follows that f^{-1} is also a symmetry and if f and g are symmetries so is fg.

Definition 2.21 gives the most general definition of a symmetry. From this it can be deduced that given any set of symmetries their closure under inverse and composition will form a group.

There are two further simplifications of the definitions of symmetries which we shall discuss, variable symmetry (Definition 2.22) and value symmetry (Definition 2.23)

Definition 2.22. A **variable symmetry** of a CSP with set of variables X is defined to be a bijective function f between the set X of variables such that given one solution, replacing the value of each variable $x \in X$ with the value of $f(x)$ also gives a solution.

Definition 2.23. A **value symmetry** if a CSP with set of variables X each with domain D is defined to be a bijective function $f : D \to D$ such that replacing the value of each variable $x \in X$ with $f(x)$ leads to another solution.

Variable and value symmetry appear easier to work with and understand than full symmetry groups, and a number of algorithms appear designed to work only on variable symmetries. It is however worth noting that the more full definition of symmetry becomes equivalent to variable symmetry if the CSP is mapped to the *binary variable representation*, where each variable X with domain D is replaced with the set $\{X_a | a \in D\}$ and $X_a = 1 \iff X = a$. Using this any method designed to work on variable symmetry can be applied to literal symmetry.

Given the symmetries of a CSP the knowledge of their existence can be used to reduce the size of the search performed. There are two major methods used to break symmetry in CP:

Static symmetry breaking methods choose before search starts a representative from each symmetric set of assignments and impose constraints so only one of them can occur during search (or at least one can occur during search for incomplete methods).

Dynamic methods alter the search procedure so once an assignment has been disregarded, no symmetric equivalent to that assignment will be explored in future.

2.4.3 Static Symmetry Breaking

The basic principle behind static symmetry breaking, first shown in [64], is to choose a privileged member from each symmetric group of assignments and place constraints which forbid any of the other symmetric assignments from occurring.

Clearly it would not be feasible to simply enumerate all the symmetric sets of assignments and place a constraint for each one. One of the most successful general methods of static symmetry breaking is based on the work of Luks, Roy and Crawford [16,55] who studied adding lexicographic constraints to the problem definition to break variable symmetries.

This method orders the variables in the CSP and then imposes constraints so that if assignments are considered as ordered under this ordering, only the lexicographically (Definition 2.24) smallest image of each assignment is allowed. This is done by imposing that for each symmetry the assignment to the variables must be lexicographically less than or equal its image under the symmetry. Example 2.8 gives a practical example of this method.

Definition 2.24. Given two equal length arrays $\overline{V}$ and $\overline{W}$, both with values from the same ordered domain, $\overline{V} \leq_{lex} \overline{W}$ if $\overline{V} = \overline{W}$ or if at the smallest value of i where $V[i]$ and $W[i]$ differ, $V[i]$ is smaller than $W[i]$.

Example 2.8. *Consider a CSP containing a* 2×2 *matrix with the symmetries that both the rows and columns can be exchanged. Below is an example assignment to this matrix, and its four symmetric equivalents under these symmetries.*

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \begin{pmatrix} 3 & 4 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 4 & 3 \end{pmatrix} \begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix}$$

The matrix $\begin{pmatrix} A & B \\ C & D \end{pmatrix}$ *can be represented by the string* $ABCD$ *and its symmetric equivalents are* $CDAB$,$BADC$ *and* $DCBA$*. This implies the constraints* $ABCD \leq_{lex} CDAB$*,* $ABCD \leq_{lex} BADC$ *and* $ABCD \leq_{lex} DCBA$ *to break the symmetries of the matrix. Only the first of the matrices above satisfies all of these constraints, showing the symmetry is broken.*

As in Example 2.8, if the variables of the matrix are labelled and considered as strings (for example read from left-to-right, top-to-bottom), constraints can be imposed ensuring one of these strings is the lexicographically least.

This system is commonly known as the *Crawford ordering*. These constraints allow exactly one representative of each symmetric class of assignments. Therefore this method is both consistent and complete. The major problem with this system is that it requires one constraint for each element of the symmetry group. A symmetry group over n variables can be as large as $n!$ and in many useful problems grows at least exponentially. It has been shown [73] that for some symmetry groups with special properties (for example abelian) it is always possible to construct polynomial sized sets of constraints. Unfortunately these special group properties seem to turn up very rarely in real problems.

One way that the problem of exponential size sets of constraints can be solved is to only use a subset of the constraints generated by the Crawford ordering. This can lead to the symmetry breaking system no longer being complete but cannot cause it to become inconsistent.

Choosing a subset of constraints has been demonstrated most effectively on matrices with symmetries of both the rows and columns [21]. Here it has been shown that the Crawford ordering generated by ordering the variables in left-to-right, top-to-bottom order contains the constraints that the rows are constrained to be in lexicographic order with the smallest on the left and the columns are constrained to be in lexicographic order with smallest at the top. This ordering shall be referred to as lex^2.

The constraints are not complete, as can be shown by the following matrix, where these two matrices are symmetric under row and column permutations but both satisfy lex^2

$$\begin{pmatrix} 1 & 2 \\ 2 & 0 \end{pmatrix} \begin{pmatrix} 0 & 2 \\ 2 & 1 \end{pmatrix}$$

lex^2 was originally proved correct by Ilya Shlyakhter [73] and independently a larger study was performed by Frisch et al. [23, 26]. Other practically useful subsets of the Crawford ordering constraints for row and column symmetry have been found [30], but it is not yet fully understood why some constraints appear to be more useful than others.

2.4.4 Dynamic Symmetry Breaking

An alternative to static symmetry breaking is dynamically breaking the symmetry during search. While there are a number of dynamic symmetry breaking algorithms with often complex implementations, the basic underlying idea is simple. A list is kept of the sub-domains of the variables at every previous node of search. At each new node, an algorithm checks if some symmetric image of this new node can be embedded in any previously checked search node. If so, then a part of the search space symmetric to this has already been searched and there is no need to do so again.

The first system which investigated this was symmetry breaking search trees [4], later extended to *SBDS* [38](Symmetry Breaking During Search) which in its simplest form imposes one constraint for each member of the symmetry group at each fail node, which forbids any symmetric equivalent from occurring. While this can be optimised slightly, it still suffers from the same problem as basic static symmetry breaking, which is that the number of added constraints is linear in the size of the symmetry group which itself often grows exponentially with regards the number of variables.

Therefore a newer system was constructed known as SBDD (Symmetry Breaking by Dominance Detection) which does not add constraints to the solver but simply takes the current node and previous failed nodes and attempts to check for the current node being contained in a symmetric version of a failed node (also known as dominated). While this system was more effective, it required a complex algorithm to detect dominance to be written for each problem.

There are any powerful algorithms for solving group-theoretic questions and a number of systems which implement these algorithms, for example GAP [31] and MAGMA [9].

CGT has been used with a high level of success in dynamic symmetry breaking with both SBDS [34] and SBDD [35], allowing generic systems which can handle groups many magnitudes larger than previous bests. Recent advances [53] have provided the CGT tools necessary to efficiently implement static symmetry breaking [48].

GAP-SBDD uses computational group theory to cope with any symmetric group it is given, making it the first of the symmetry breaking systems which could be used by the untrained user, including help in constructing the group itself [62]. In a similar way to $STAB$, the runtime of GAP-SBDD can be exponential at each search node. In an experiment of a problem with a search group of size 10^{30}, GAP-SBDD took over 25 seconds at each search node, whereas without symmetry breaking it is possible to search hundreds of thousands of nodes a second. Despite this, as the symmetry group is so large the search is still faster.

While the implementation details of these, and other dynamic symmetry breaking systems vary the underlying principle is to record which areas of the search space have been searched previously, and then avoid searching areas symmetric to them when they are later reached in the search space.

2.4.5 Comparison

There are advantages and disadvantages to the two methods of symmetry breaking (static and dynamic). Static is typically much faster at each node of the search tree as it requires less work. However with static symmetry breaking it is possible for the solver to be prevented going down a branch of the search tree that contains a solution because it is forbidden by the symmetry breaking constraints. Dynamic symmetry breaking does not have this problem as it will only block parts of the search tree symmetric to ones already investigated. However it appears that as long as the variable ordering is chosen with care then static symmetry breaking can be competitive (it is possible to choose variable orderings which cause static symmetry breaking to perform many times worse, however as the variables in the problem are symmetric in some way it is arguable that the choice of variable ordering can be up to a point decided by the symmetry breaking constraints).

Dynamic symmetry breaking methods are usually computationally more expensive than the static symmetry breaking, and require extra support from the solver itself. One of the most interesting parts of static symmetry breaking is that it can lead to extra useful implied constraints. This will be explored in depth later.

2.5 Conclusion

This chapter has covered a large number of basic mathematical concepts and a brief introduction to constraint programming. The contents of this chapter are the building blocks on which all of modern constraint programming is built. Chapter 3 will look at those areas of the literature which more directly relate to this thesis and the holes left in the existing research which this thesis investigates.

Chapter 3

Related Work

How to represent high-level variables in CSPs is something which is handled, at least implicitly, in the majority of papers on constraint programming. In particular, every paper which contains a constraint model of a pre-existing problem has to make a choice on how to represent the original problem as a CSP. In the vast majority of these papers however, which representations were considered, and how the one which was used in the experiments was chosen, is neglected.

This chapter discusses a number of areas of research which have investigated how to generate and compare different CSP models of the same problem and the holes in this existing research which are studied in this thesis.

3.1 Comparisons of Representations

One vital part of comparing different CSP models of the same problem is providing a method of defining how the different CSP models relate to each other. One of the first such theories was equivalence of CSPs, first discussed by Rossi et al. [69]. This work discussed ways in which CSPs can be considered equivalent. The first naive definition of equivalent CSPs is that two CSPs share the same sets of variables, domains and solutions. A second, more useful definition is given in Definition 3.1, using the idea of reducibility.

Definition 3.1. A CSP $P_1 = \langle V_1, D_1, C_1 \rangle$ is reducible to a CSP $P_2 = \langle V_2, D_2, C_2 \rangle$ if there is an function f mapping V_1 to V_2 and a function g_v for each $v \in V_1$ mapping $dom(v)$ to $dom(f(v))$, such that for any $v \in V_1$ and $d \in dom(v)$, the value of v in each solution of P_1 is d whenever the value of $f(v)$ in the solution of P_2 is $f_v(d)$.

P_1 and P_2 are equivalent if P_1 is reducible to P_2 and P_2 is reducible to P_1

While equivalence provides a good start, it fails to capture many examples of different representations of different problems which often arise in CP. An extension of equivalence came from the study of viewpoints (Definition 3.2), first introduced by Geelen [33].

Definition 3.2. A **viewpoint** of a problem P is defined as a pair $\langle X, D_X \rangle$, where $X = \{x_1, \ldots, x_n\}$ is a set of variables, and D_X is a set containing, for each $x \in X$, an associated domain $D_X(x)$ giving the possible values for x. Each assignment to $\langle X, D_X \rangle$ must be related to an assignment to P. A set of constraints on a viewpoint is valid if they ensure the solutions to the viewpoint exactly represent the solutions to P.

Viewpoints provide a much more general framework than equivalence. Example 3.1 gives an example of a common viewpoint. The major problem of viewpoints is that they are too general, as any viewpoint can represent any problem with no solutions if constraints are placed such that all solutions are forbidden, a problem recognised by the authors.

Example 3.1. *Consider a CSP which requires finding a permutation of the set* $\{1, \ldots, n\}$*. One obvious viewpoint of this is as an array* $\overline{V}$ *of length* n *of variables with domain* $\{1, \ldots, n\}$*, where* $V[i]$ *represents the* i^{th} *element of the permutation.*

Another viewpoint of this problem is another array $\overline{W}$ *of length* n *of variables with domain* $\{1, \ldots, n\}$*, where* $W[i]$ *gives the position of* i *in the permutation. These two viewpoints are connected by the condition that* $V[i] = j$ *if and only if* $W[j] = i$*.*

Despite the limitations of viewpoints, they can still be used to usefully compare different models of the same problem. Law, Lee and others [13, 51, 52] have compared different viewpoints of the same problem, in particular the viewpoints given in Example 3.1. They showed significant improvements could be gained in CSPs with only binary constraints when multiple viewpoints were used in the same problem. In general, when multiple viewpoints or models are used in the same problem, constraint must be imposed on them to ensure they represent the same solution. Such constraints are called channelling constraints and are defined in Definition 3.3. This definition is purposefully a little vague, as the exact form of channelling constraints can vary depending on how the model is constructed.

Definition 3.3. Given two sets of variables $\overline{V}$ and $\overline{W}$ and a pair of functions f and g which map assignments to $\overline{V}$ and $\overline{W}$ into a set D, where $f(\overline{v}) = g(\overline{w})$ means $\overline{v}$ and $\overline{w}$ represent the same assignment, then the constraint $f(\overline{V}) = g(\overline{W})$, or any constraint logically equivalent to it is a **channelling constraint** between $\overline{V}$ and $\overline{W}$.

This thesis addresses two major problems from this work in viewpoints. Firstly by defining representations independently of the constraints of the problem structure is much better preserved, avoiding the problem of all viewpoints with no solutions being equivalent. Secondly, Section 5.3 will show how many of the apparent differences between different viewpoints are due to artificial limitations, for example requiring binary constraints, which can be easily removed.

There have been a number of previous papers which have carefully studied a selection of representational choices, and given both theory and experimental evidence to help both modellers and automated modelling systems to make an optimal choice.

One place where this kind of comparison has been made previously is the thesis of Brahim Hnich [45], which studied alternative representations of different families of functions.

Hnich looked at a number of function representations, some of which are appropriate only for particular types of functions, such as total or bijective. Further, some representations performed much better for particular types of functions. On these representations he compared the implementations of a number of constraints, including finding the inverse, range and domain of a function.

This thesis expands on and differs from this earlier work in a number of ways. While providing a number of representations of functions, Hnich did not try to provide a unifying framework which included these and other representations found in constraint programming. The comparisons in Hnich's work concentrated on many representations which are viewed as equivalent in this thesis (Definition 5.2 on page 75) and studied how different propagators compared on these models. This thesis takes a different route, ignoring issues to do with different levels of propagation while allowing the comparison of more disparate models.

3.2 Refining Specifications

The main aim of this thesis is to investigate the choices generated in choosing between different representations of the high-level variables. A separate issue is how to automatically use these representations and automatically produce CSP models of a high-level specification of a problem.

OPL [42] provides a simple language for modelling CSPs, which allows many problems to be compactly specified. It is however limited as it only provides integer and enumerated variables. ESRA [22] is a language which aims to build on OPL, and provides support for variables which represent sets and relations. ESRA is not intended to be solved directly, but instead specifications in ESRA are refined to specifications in OPL by simplifying variables and constraints into a form which OPL will accept.

Similarly, NP-Spec [11] allows the specification of NP-complete problem in a subset of existential second order logic. NP-Spec provides a small number of high-level types, such as sets and partitions of integers, which are refined down into simpler types. Also like ESRA, it only provides a single possible refinement for each type, and each constraint which can be defined on it.

F [45] is a language which provides support for different types of functions on integers. F is refined by a system called Fiona into a subset of OPL. The major advantage of Fiona over other systems is that it knows of a number of different representations of function variables and is able in some cases to choose automatically which should be used. F supports a number of special classes of functions, such as total or bijective. Furthermore, if the same variable can be refined multiple different ways in the same CSP, then the different models are connected by constraints which ensure the different models achieve the same solution.

All the systems of refinement discussed so far this section only provide support for a small fixed range of high-level constructs. The ESSENCE [27] language does not have this limitation. ESSENCE supports arbitrarily complex types, for example sets of sets, or functions of partitions. ESSENCE is refined by a system called CONJURE [28], which uses a series of recursive rules to refine an ESSENCE specification into a subset of the language suitable for giving the existing constraint solvers. CONJURE can produce different models of the same variable like Fiona.

The major current limitation of CONJURE is that it produces a large number of refinements of the same specification and does not make any decisions on which of these would be best to use. The aim of this thesis is to work towards filling this gap in automated modelling, producing a framework which can be used to choose between different representational choices.

3.2.1 Program Reformulation

While the field of CSP reformulation is quite new, there is a much larger body of work on the reformulation of general computer programs, and some of these systems have been used to create efficient CSP implementations. Therefore it is informative to look at how these systems have evolved over time and how useful they can be in implementing refinement of specifications in CP.

KIDS [1] is one such system. It allows the user to specify a program in a high-level language and a set of possible re-formulations with pre-conditions (of which a very large number are included in the system). The system then guides the user through the reformulation of their program by giving a selection of options on how reformulation can progress, each of which is checked by KIDS to ensure it will not alter the correct functioning of the program. KIDS is built on a very-high-level language REFINE [2] which supports many high-level constructs and was designed to help support program reformulation and refinement. Using KIDS usually follows the following process.

1. *Develop a domain theory* for the problem, defining appropriate types and functions and high-level reasoning about the defined functions, such as information about distributivity and monotonicity which support design and optimisation. Obviously this can be done once if many programs are being written in a similar field.

2. *Create a specification* of the problem stated in terms of the underlying design theory

3. *Apply a design tactic* such as divide-and-conquer, global search or local search

4. *Apply optimisations* to the resulting correct (assuming we have the domain theory and specification correct) but inefficient program. KIDS includes many optimisation techniques such as simplification, partial evaluation and finite differencing.

5. *Apply data type refinements* to the high level data constructs in the program, using a set of data type refinement rules

6. *Compile* the resulting code (which is in REFINE) to an executable form (LISP).

KIDS has been used directly to generate efficient implementations of constraint satisfaction programs, for example by Westwood and Smith [77] who took a simple specification of the 8 queens problem and a branching search algorithm and generated an efficient implementation of the program using KIDS. This work was effective and also allowed re-use of an existing mature reformulation and refinement program. For the solving of CSPs it would however be useful to move at least some of the optimisation phase to act on the specification. CSPs have a number of useful properties which a more general system may not be able to effectively make use of. For example if some algorithm implements a correct pruning algorithm for a constraint then it can always be replaced with another correct pruning algorithm which may be stronger or weaker and the final result of the program will be the same. Also the order in which the pruning algorithm is applied does not affect the final result (although it may of course effect the running time). The existence of systems like KIDS does provide encouragement to the idea that an efficient refinement / reformulator can be designed specifically for solving CSPs.

One of the major weaknesses of the KIDS system is that while it is capable of suggesting possibly useful re-formulations via heuristics (for example, finding small but non-trivial transformations) and making sure that any reformulation does not effect the correct execution of the problem, it does not however attempt to suggest which re-formulations should be applied. Some of the re-formulations it performs can be shown to never reduce the execution speed of a program (such as extracting constant expressions from loops or not evaluating expressions unless they will be used) but with others the effects can be more difficult to predict. Therefore the decision whether to apply a reformulation is always left up to the judgement of the user. In the field of constraint programming, this thesis aims to develop a framework in which choices can be made about modelling independent of users.

An example of a system which makes use of alternative formulations of high level objects is the SETL [72] language, a language which supports sets and operations upon them in the basic language. SETL is capable of considering different representations of a set depending on the operations which will be performed upon it. This appears to be the kind of thing which would be of interest in the search for automatic reformulations of constraint satisfaction problems, but unfortunately the problems turn out to be more separate than they at first appear. When considering how an object should be represented in a general program, the decisions are based upon the problem of size and speed. While these are also important in constraints, some of the most important parts of constraint programming do not map well to a software refinement framework. On one hand CSP can be seen as very trivial, as a CSP specification has no state or loops. In this case the tools of software refinement, while useful as a guide are designed for a much more general system and are over-complex. If instead a CSP along with a solver are viewed as a single unit to be refined, it is difficult to express changes which can alter the execution of the program to the point where it is unrecognisable, yet still produce a set of solutions which can be viewed as equivalent[1]. Previous work on data refinement can give important pointers on how to actually go about generating low level models based on high level specifications, and also how to implement these low level models as efficiently as possible. However the problem of choosing between these representations is one where new research will have to be performed, although partly inspired by the ideas contained in data refinement. This thesis builds a framework in which some of these decisions can be made.

3.3 Implementing High-Level Representations

In modern constraint solvers many constraints have special propagation algorithms composed for them, as discussed in Section 2.3.5. There is a large body of research on designing propagators for constraints on different representations of a number of high-level variable types. One of the earliest and most influential was Conjunto [39], which implemented a number of common constraints on the *occurrence* (Definition 4.19 on page 67) representation of sets, providing an efficient and simple to use implementation of many common set constraints. This work has since extended into other domains, such as multisets [49, 76] and graphs [18].

Toby Walsh et. al. [6, 76] discuss how to implement many constraints on the *occurrence* representation of sets and multisets, both with and without a fixed size on the set. Walsh et al. also show for a number of constraints that checking if a set of sub-domains is GAC is NP-complete. This work has been

[1]see for example the discussion on (multi)sets in section 4.1

extended by Sadler [71] to consider more complex set representations, including the lexicographic representation (Definition 4.25 on page 71).

While this work has produced increasingly powerful representations which can handle a number of constraints, they are difficult to compare. In many cases new representations are created by joining extra information to an existing representation, or producing more powerful propagators. Also in most cases the major aim of these papers is to either prove or disprove that GAC propagation can be achieved efficiently. This thesis instead compares these representations to see how well they approximate the best possible representation, providing a different and complimentary categorisation of the representations.

One special case of implementing set variables is using reduced ordered binary decision diagrams (ROBDDs), by Stuckey et al. [50]. These take a different route, implementing the representation which allows all possible sub-domains of each variable and therefore requiring worst-case exponential time and space to perform propagation. ROBDDs have a number of advantages, including providing an efficient practical framework to easily synthesise constraints together and combine a number of different set representations. It will be shown in this thesis that ROBDDs produce the smallest search tree of all set representations on all problems, but usually take much longer per node, often many orders of magnitude slower in practice. In exactly what cases this trade-off should be made is not yet understood, but this thesis goes some way towards providing an answer to this question.

While ignoring how to practically implement constraints clearly means the results of this thesis must be used with care, it allows a number of powerful frameworks to be built, which can then be applied to these existing and future implementations, and show where improvements can be made.

3.4 Conclusion

This chapter has presented the current state of the art in representations in constraint programming. One common theme throughout this section is lack of study of variables in isolation from the constraints placed on them and the comparison of representations which do not have a simple viewpoint relationship. The later chapters of this thesis provide and investigate a general framework by which all representations can be usefully compared and contrasted, allowing new insights into modelling problems as constraint problems.

Chapter 4

Introduction to Representations

Section 2.3.3 gave a simplistic introduction to backtracking propagating constraint solvers. One important missing point from this is exactly how the domains of the variables are represented and stored during search. It is only by removing values from the domains of variables that constraint propagators can affect search and communicate with each other. Therefore the way in which these domains are stored, used and changed is fundamental to the efficient operation of a constraint solver. The aim of this thesis is to study these different representations and show how the differences between them affect search.

This chapter examines how the representation of high-level variables such as sets and multisets is implemented in modern CP solvers and builds a theory which can represent existing representations. This chapter also shows how the presented theory avoids a number of problems which can arise when trying to build a theory of how constraint solvers store representations. Finally, a number of practical example of existing representations will be shown to fit into this theory.

4.1 Introduction and Examples

This section provides a number of motivating examples which shows the major features of representations as provided by many modern CP solvers. The important ideas of these will then be extracted and used to define a general theory of representations.

Example 4.1. *Consider a variable v with domain $\{1, \dots, 10\}$, where the solver being used stores only sub-domains of v which are complete ranges of values.*

These can be represented by pairs of integers $\langle a, b\rangle$ where $a < b$ and $\langle a, b\rangle$ represents the sub-domain $\{a, a+1, \ldots, b\}$. In the general case of a sub-domain of $\{1, \ldots, n\}$, this reduces the number of allowed sub-domains of v from the original 2^n to less than n^2. Clearly any representation must require at least n states, for each of the possible assignments, so this new representation is both exponentially smaller than the naive one and close to the theoretical minimum.

The following list provides two small examples of using this new "range" representation in practice:

1. *From the initial domain of v, the constraint $\mathbf{v \le 5}$ allows the set of assignments $\{1, \ldots, 5\}$, which can be represented exactly by $\langle 1, 5\rangle$. Also imposing the constraint $\mathbf{v > 2}$ leaves $\{3, 4, 5\}$, which can be represented exactly by $\langle 3, 5\rangle$.*

2. *The smallest range of integers which contains all values allowed by constraint* **v *is even*** *is $\langle 2, 10\rangle$. This also represents the set of assignments $\{3, 5, 7, 9\}$, which are disallowed by the constraint.*

Example 4.1 demonstrates one common way in which integer variables are represented in a CSP solver, allowing only those sub-domains which form a complete range of integers. As the example shows, in some cases the result of propagating a constraint can be exactly represented by a range, whereas in other cases there may not be a range which exactly represents the allowed assignments. Clearly propagation can not be permitted to remove values which are allowed by the constraint, so instead the range generated by propagation must contain some values which do not satisfy the constraint. This does not make the propagator in any way invalid, but means that other constraints will not be passed as much information as possible, which can increase the size of search.

For some constraints which occur over integer variables, after propagation it is always possible to exactly represent the values remaining as a range. In these cases representing the allowed domain as a range improves the performance of the solver, while not increasing the size of search. In other cases, a trade-off is made between increased speed and lower memory usage at each node of search while causing larger search spaces.

Example 4.2. *Consider a set variable s whose domain is $\mathbb{P}(\{1, \ldots, 10\})$. One common compact representation of some of the sub-domains of s is with an array V of length 10 of Boolean variables. The i^{th} element of V represents if i is in the* *set or not. There are $2^{2^{10}}$ possible sub-domains of s. There are 3 possible sub-domains of a Boolean variable (unassigned, assigned* True *or assigned* False*), so only 3^{10} representation states of the Boolean array. As there are 2^{10} possible assignments to the set, the Boolean array takes exponentially less space than the naive representation, and is close to the theoretically smallest.*

The following list gives two small practical examples of using a Boolean array to represent sub-domains of a set variable in practice.

1. *The constraints $\mathbf{2} \in \mathbf{s}$ and $\mathbf{2} \notin \mathbf{s}$ map to the constraints $\mathbf{V}[\mathbf{2}] = \text{True}$ and $\mathbf{V}[\mathbf{2}] = \text{False}$. Propagating these two constraints fully will empty the domain of $V[2]$, and therefore lead to failure.*
2. *The constraints $|\mathbf{s}| = \mathbf{5}$ and $|\mathbf{s}| = \mathbf{6}$ map to the constraints $\mathbf{sum}(\mathbf{V}) = \mathbf{5}$ and $\mathbf{sum}(\mathbf{V}) = \mathbf{6}$. From the complete domain of V, every assignment to every variable is part of a solution to both of these constraints, although of course not both constraints at the same time, and therefore no propagation will occur if the constraints are propagated independently.*

Example 4.2 gives a more complicated representation example, where a set variable is represented with an array of Boolean variables. This differs from Example 4.1 as here the result of representing a set by an array of Boolean variables could be considered as generating a new CSP, rather than deciding how a solver should internally represent an already given CSP. Of course, this could also still be considered as a mechanism for representing only some possible sub-domains, internal to the solver.

The first pair of constraints in Example 4.2, once propagated, lead to a domain wipeout. The second pair of constraints would lead to a domain wipeout if considered on the original set variable but do not lead to any propagation when the set is represented as an array of Boolean variables.

Example 4.3 gives another method of representing sets.

Example 4.3. *Consider a variable X whose domain is all sets of size 2 with values drawn from $\{1, 2, 3\}$. This can be represented by a pair of integer variables $\langle V_1, V_2 \rangle$, each with domain $\{1, 2, 3\}$ and the constraint $V_1 \neq V_2$. Assignments to X, V_1 and V_2 are related by the constraint $X = \{V_1, V_2\}$. The following list shows two small examples of using this representation in practice.*

1. *Consider the constraint $\mathbf{2} \notin \mathbf{X}$. This would map to the constraint $V_1 \neq 2 \wedge V_2 \neq 2$. If this constraint is propagated, the only remaining assignment*

to X represented by V_1 and V_2 is $\{1, 3\}$ and therefore all possible elements of X which do not satisfy the constraint have been removed.

2. *Consider the constraint $\mathbf{2} \in \mathbf{X}$. This is represent on V_1 and V_2 by the constraint $\mathbf{V_1} = \mathbf{2} \vee \mathbf{V_2} = \mathbf{2}$. All possible assignments to both V_1 and V_2 are part of a solution to this constraint, as given any assignment to V_1, V_2 could be assigned 2, and vice veras. Therefore it is not possible for propagation to remove any literals. However, the assignments $V_1 = 1, V_2 = 3$ and $V_1 = 3, V_1 = 1$ both represent $X = \{1, 3\}$, which does not satisfy the constraint.*

   The sub-domains $V_1 \in \{1, 2\}, V_2 \in \{2, 3\}$ represents the sub-domain $x_1 \in \{\ \{1, 3\}, \{2, 3\}, \{1, 2\}\ \}$. The sub-domains $W_1 \in \{1, 3\}, W_2 \in \{2\}$ represents the sub-domain $x_2 \in \{\{1, 2\}, \{2, 3\}\}$. While $x_2 \subset x_1$, it is not the case that the W_i can be reached from the V_i by propagation. This shows that even when two particular sub-domains can be represented, it may not be possible to reach them in search.

Example 4.3 demonstrates three important features which can complicate representations. In all three of these examples, some constraints propagate as much when considered on the representation as the original domain, while others perform worse.

Examples 4.1, 4.2 and 4.3 each show some common problems which arise in representational theory. In each case the number of sub-domains permitted is massively reduced. Some constraints are unaffected by this reduction as the results of their propagation are one of the array of allowed sub-domains. Other constraints do not fit so well with the new representations, and are forced to use a representational state which also represents assignments which are forbidden by the constraints. The aim of this thesis is to investigate these problems in depth and as far as possible give a categorisation of how different representations will perform on different CSPs.

There is a large range of practical and theoretical problems which contain set and multiset variables in their most natural specifications. In this thesis two example problems are considered. The Social Golfers problem (Definition 4.1, problem 10 at www.csplib.org) [5, 41, 74] is a frequently studied problem in CP. It is also interesting in the context of this thesis due to the nested nature of the problem's variable. The BIBD problem (Definition 4.2, problem 28 at www.csplib.org) are two problems which have been modelled as CP problems and used as examples in many previous CP papers. They will be used as running examples throughout this thesis.

Definition 4.1. The $\langle g, p, w\rangle$ social golfers problem requires finding a schedule for g golfers to play on each of w weeks split into p periods in each week, where no two golfers play together in more than one week. It is only defined in the cases where $\frac{g}{p}$ is an integer. A more formal specification of the problem is:

- Given a set G of size g.
- Find a multiset[1] *Sched* of size w, where each element of *Sched* partitions G into p equal sized sets.
- Such that no two elements of G are in the same partition for more than one element of *Sched*.

Definition 4.2. A $\langle v, b, r, k, \lambda\rangle$ BIBD requires, given a set S of size v, finding b sets of size k where each element of S occurs in r sets and the intersection of each pair of sets is size λ. A more formal specification of the problem is:

- Given a set S of size v.
- Find a set $BIBD$ of size b, where each element of $BIBD$ is a subset of S of size r.
- Such that each element of S occurs in k elements of $BIBD$ and the size of the intersection of any pair of elements of $BIBD$ is λ.

4.2 Theory of Representations

Section 4.1 showed some examples of representations. This section will outline a theory which can be used to describe both these and other representations. Based on the Examples seen earlier in this chapter, a good basis for a representation is a set of sub-domains with a partial ordering (Definition 2.5 on page 16), where the partial ordering shows which new sub-domains can be reached from a given sub-domain during propagation.

Example 4.4 shows how considering a representation as an arbitrary set of sub-domains can lead to problems. In this example, the representation chosen

[1]Except in degenerate cases, this can be proved to be a set. However, this is an implied constraint rather than part of the problem.

leads to a problem, where the order in which constraints are propagated gives different results. This creates a number of difficulties and does not line up with the behaviour of both the theory and practice of constraint solvers. In particular, this also means the naively defined propagators would not be monotonic, which is one of the conditions which a propagator (Definition 2.18) must satisfy.

Example 4.4. *Consider an integer variable X with domain $D = \{1,2,3,4\}$. The "Stupid Solver" representation can represent all sub-domains of X which are* not *of size 2. Consider propagating the two constraints* $\mathbf{X \geq 2}$ *and* $\mathbf{X \leq 3}$ *to the maximum possible extent. There are two cases.*

1. *First propagate* $\mathbf{X \geq 2}$*, leaving the sub-domain $\{2,3,4\}$. Propagating* $\mathbf{X \leq 3}$ *would remove 4, leaving $\{2,3\}$, but this cannot be represented by the solver.*
2. *First propagate* $\mathbf{X \leq 3}$*, leaving the sub-domain $\{1,2,3\}$. Propagating* $\mathbf{X \geq 2}$ *would remove 2, leaving $\{2,3\}$, but this cannot be represented by the solver.*

Therefore the order in which constraints are propagated leads to different results.

While it may be possible to define representations in some fashion which allows representations such as those in Example 4.4, they would not fit well with the traditional algorithms and theorems used in CP and therefore a more complex definition will be used in this thesis, which more accurately models existing representation and their use in constraint solvers.

The major reason that Example 4.4 fails to be a representation is that given two propagators, the *fixed point* reached by applying both until propagation stops differs depending on the order in which they are applied. *Lattices*, given in Definition 4.3 are a special kind of partially ordered sets which exactly impose the condition that the fixed point is independent of the order in which the propagators are applied. Example 4.5 gives an example lattice, inspired by Example 4.1 on page 45.

Definition 4.3. Given a partially ordered set $\langle S, \leq \rangle$, the lowest upper bound of two elements $s, t \in S$, denoted $lub(s,t)$, is the member of S satisfying the following two conditions, where such a member exists:

1. $\forall s,t \in S.\ s \leq lub(s,t) \wedge t \leq lub(s,t)$
2. $\forall s,t,u \in S.\ s \leq u \wedge t \leq u \implies lub(s,t) \leq u$

The greatest lower bound *glb* of two members of S is defined similarly. A **lattice-ordered set** S is a partially ordered set where all pairs of elements $s, t \in S$ have both a lowest upper bound and greatest lower bound.

Example 4.5. *Consider the set* $S = \{[a..b] | 1 \leq a \leq b \leq n\}$*, where* $[a..b]$ *represents the set* $\{a, a+1, \ldots, b\}$*. One partial ordering on this set is inclusion, that is* $[a..b] \leq [c..d] \iff a \leq c \wedge d \leq b$*. It is easy to check this satisfies the requirement of being a partial ordering. Furthermore, every pair of ranges in this definition has a lowest upper bound, given by* $lub([a..b], [c..d]) = [min(a, c)..max(b, d)]$*.*

Not all pairs have a greatest lower bound however. If $\emptyset$*, representing the empty range, is added to* S *however, then they do. The greatest lower bound is given by* $glb([a..b], [c..d]) = [max(a, c)..min(b, d)]$*, where this range denotes* $\emptyset$ *if* $max(a, c) > min(b, d)$*.*

There are a number of useful mathematical properties of lattices which will be used in this thesis. These are summarised in Lemma 4.1.

Lemma 4.1. *The following properties are true of all finite lattices* $\langle S, \leq \rangle$*.*

1. *There is exactly one lowest upper bound and greatest lower bound of each pair of elements of* S*.*
2. *There exists a value* $S^{\Uparrow}$ *in* S *such that* $S^{\Uparrow}$ *is greater than every other element of* S*.*
3. *There exists a value* $S^{\Downarrow}$ *in* S *such that* $S^{\Downarrow}$ *is smaller than every other element of* S*.*

Proof.

1. If g is a greatest lower bound of a and b, then by definition any h such that $h \leq a$ and $h \leq b$ must satisfy $h \leq g$. If h was also a greatest lower bound of a and b, then similarly $g \leq h$ so $g = h$. The greatest lower bound follows identically.
2. If the list of elements of S are labelled s_1 to s_n, then consider the value of the expression $lub(s_1, lub(s_2, \ldots, lub(s_{n-1}, s_n) \ldots))$. By definition, this must be greater than or equal to all the s_i, and also still in the lattice. Therefore it gives a maximum element, larger than every other value.

3. Proof follows similarly to part 2.

□

Some simple examples of representations, including Example 4.2 on page 46 and Example 4.1 on page 45, can be defined using only a lattice of sub-domains. However, more complex examples like Example 4.3 on page 47 show how it is not possible to simply define a representation as a set of sub-domains. Instead it is necessary to define the states of the representation (in the case of Example 4.3 these are a sub-domain of each variable) independently of the sub-domains they represent. This condition is encapsulated in Definition 4.4, along with a number of other requirements.

Definition 4.4. A **representation** of a domain D is a pair $\langle R, f\rangle$, where R is a lattice and f is a homomorphism from R to the lattice of sub-sets $\langle \mathbb{P}(D), \subseteq\rangle$. $\langle R, f\rangle$ must satisfy these three conditions:

1. $f(R^{\Uparrow}) = D$.
   The top state allows all assignments.
2. $f(R^{\Downarrow}) = \emptyset$.
   The bottom state allows no assignments.
3. $\forall r \in R.\ \forall a \in f(r).\ \exists r' \in R.\ (f(r') = \{a\} \land r' \leq r)$.
   If a state r contains the assignment a, a is reachable from r.

An element of R is known as a **state** of the representation, and given a state r, the sub-domain that state **represents** is $f(r)$. Given $r_1, r_2 \in R$, r_1 is **reachable** from r_2 if $r_1 \leq r_2$. The set of assignments **allowed by** $r \in R$ is $f(r)$. A state $r \in R$ is an **assignment** if $|f(r)| = 1$.

Definition 4.4 gives a number of requirements on representations, which are necessary so a representation can be used in a tree search. Clearly there must be a state which can be used as the start of search. The top node of the lattice is the node which will represent the subset of the assignments represented at every other node, so this one is used. Similarly there must be a node which represents no assignments, else there is no way for propagators to cause failure. The bottom node of the lattice must represent the intersection of the subset of the assignments represented at all other nodes, so if any node of the lattice represents no assignments the bottom one will.

The third condition encapsulates the idea that a representation state is a set of possible assignments. Without this condition, it would be possible for a representational state to contain some value a, but no state reachable from it be the assignment a. It would be very difficult to ensure that all assignments were checked without this condition. One condition which may appear to be missing is that there must be a representation state which represents each possible assignment. This is however implied by point 1 and 3 in the definition.

There are a number of interesting families of representations which frequently occur in this thesis. Examples 4.1 and 4.2 both gave examples of representations where the ordering on the elements of the representation is exactly the ordering on the sub-domains they represent and therefore the representation can be defined purely in terms of the set of sub-domains which are represented. Not all representations satisfy this requirement, for instance Example 4.3 does not. This class of representations, known as simple, will be defined specially as they occur frequently and some important theorems hold only for simple representations. Simple representations are formally defined in Definition 4.5. Lemma 4.2 formalises the representation given in Example 4.2, and proves it is simple.

Definition 4.5. A *simple representation* of a domain D is a representation $\langle R, f\rangle$ which satisfies the condition $\forall r_1, r_2 \in R.\ f(r_1) \subset f(r_2) \implies r_1 < r_2$.

Lemma 4.2. *Consider representing a variable with domain $\{l, \ldots, u\}$ by the ranges of integers $\{a, a+1, \ldots, b\}$ for all $1 \leq a \leq b \leq n$. The pair $\langle R, f\rangle$, where:*

- *$R = \{[a, b] | l \leq a \leq b \leq u\} \cup \{\emptyset\}$ with the ordering given by:*
  $\forall [a_1, b_1] \in R.\ [a_1, b_1] > \emptyset$
  $\forall [a_1, b_1], [a_2, b_2] \in R.\ (a_1 \leq a_2 \wedge b_2 \leq b_1) \iff [a_1, b_1] \geq [a_2, b_2]$
- *f is defined by $f([a, b]) = \{x | a \leq x \leq b\}$ and $f(\emptyset) = \emptyset$*

is a simple representation of this domain.

Proof. To show R is a lattice, it must be shown that for any pair of values $r_1, r_2 \in R$, both $lub(r_1, r_2)$ and $glb(r_1, r_2)$ are well-defined. Given $[a_1, b_1]$ and $[a_2, b_2]$ in R, the smallest range which contains both these ranges is $[min(a_1, a_2), max(b_1, b_2)]$, and further, clearly any range which contains both the ranges must contain this range. Defining the glb follows similarly. Finally, $lub([a, b], \emptyset) = [a, b]$ and $glb([a, b], \emptyset) = \emptyset$.

To check $\langle R, f\rangle$ is a simple representation, each of the four conditions a simple representation must satisfy from Definition 4.4 must be checked. The first 3 are easily checked, as $f(R^{\Uparrow}) = f([l,u]) = \{l, l+1, \ldots, u\}$, $f(R^{\Downarrow}) = f(\emptyset) = \emptyset$ and $\forall l \leq i \leq u.\ f([i,i]) = i$. The final condition required for a representation and the condition of being a simple representation can be checked at the same time, as ignoring the case of $\emptyset$, they together require $[a,b] < [c,d] \iff \{i|a \leq i \leq b\} \subset \{i|c \leq i \leq d\}$, which is true, and similarily $\emptyset < r \iff \emptyset \subset f(r)$ is true. $\square$

For any domain, the **complete representation** is the simple representation which allows all possible sub-domains of a variable. The complete representation comes up frequently, as it is effectively how most constraint solvers implement simple variable types, such as integers, and also because Theorem 5.3 on page 77 proves that at least for static variable orderings it always leads to the smallest possible search size.

4.3 Variable Representations

One very common pattern in the representations in common usage is representations which arise by replacing one variable with a number of other variables with smaller domain. Both Example 4.2 and 4.3 gives examples of this kind of representation. These differ significantly from other representations as the result of applying them can be seen as another CSP, whose variables could also be replaced by representations. For this reason a specialised theory of these kind of representations, called *variable representations* has been divised. Definition 4.6 gives a general definition of variable representations.

Definition 4.6. A **variable representation** of a variable X is a pair $\langle \overline{V}, g\rangle$, where $\overline{V}$ and g are defined as follows:

$\overline{V}$: An array of variables.

g: A surjective function from $dom(V[1]) \times dom(V[2]) \times \cdots \times dom(V[|\overline{V}|])$ to $dom(X)$.

Definition 4.7. Given a variable representation $\langle \overline{V}, g\rangle$ of a variable X, the representation **induced** from $\langle \overline{V}, g\rangle$ is a representation $\langle R, f\rangle$, where R and f are defined as:

R is a lattice whose elements are all arrays of sub-domains of elements of V, of more precisely the set $\{\langle s_1, s_2, \ldots, s_{|\overline{V}|}\rangle | \ s_i \subseteq dom(V[i])\}$. The ordering on R is $\forall\{r_1, r_2\} \subseteq R$. $(r_1 < r_2 \iff \forall i.\ r_1[i] \subseteq r_2[i])$.

f maps an array of sub-domains to a set of assignments of X by generating every assignment the sub-domains represent and applying g to each of them. The exact mapping is given by $f(\overline{r}) = \{g(a_1, a_2, \ldots, a_n) \mid \forall i.\ a_i \in r[i]\}$.

Definition 4.6 defines variable representations and simple variable representations in terms of replacing one variable with an array of variables. Definition 4.7 shows how to induce a representation from a variable representation. Lemma 4.3 shows that the representation induced using Definition 4.7 are indeed correct representations and simple representations in the traditional sense.

Lemma 4.3. *The representation $\langle R, f\rangle$ induced by a variable representation $\langle \overline{V}, g\rangle$ of a domain D is indeed a representation. $\langle R, f\rangle$ is simple if and only if g is injective.*

Proof. Given $r_1 = \langle x_1, \ldots, x_n\rangle$ and $r_2 = \langle y_1, \ldots, y_n\rangle$ in R, as the $glb(r_1[i], r_2[i]) = x_i \cup y_i$, then $glb(r_1, r_2) = \langle x_1 \cup y_1, x_2 \cup y_2, \ldots, x_n \cup y_n\rangle$. The *lub* is defined similarly and therefore R is a lattice. To show $\langle R, f\rangle$ is a representation, it therefore suffices to check the four conditions given in Definition 4.4.

1. As g is surjective, $\forall d \in D$ there exists an assignment v_d of V such that $g(v_d) = d$. Therefore $f(R^{\Uparrow}) = D$.
2. $f(R^{\Downarrow}) = \emptyset$.
3. $\forall d \in D.\ \exists r \in R.\ f(r) = \{d\}$.
4. $\forall r_1, r_2 \in R.\ r_1 < r_2 \implies f(r_1) \subset f(r_2)$.

If g is not injective, then there exists two assignments $\overline{v_1}$, $\overline{v_2}$ to $\overline{V}$ such that $g(\overline{v_1}) = g(\overline{v_2})$. Therefore there must be two states of R representing just these two assignments, which will be incomparable but map to equal sub-domains, so $\langle R, f\rangle$ cannot be simple.

If g is injective, then given $r_1, r_2 \in R$, if $f(r_1) \subset f(r_2)$, then the set of assignments to V allowed by r_1 must be a subset of those allowed by r_2, so $r_1 < r_2$, so $\langle R, f\rangle$ is simple. □

As stated previously, the main advantage of variable representations is that they map a CSP to another CSP, which can then either be given to a CSP solver or have further representation choices applied to it. Definition 4.8 shows how a variable in a constraint can be replaced by a variable representation.

Definition 4.8. Given a constraint C and a variable representation $\langle \overline{V}, f \rangle$ of a variable $X \in \mathit{scope}(C)$, then *replacing X in C by $\langle \overline{V}, f \rangle$* is defined as follows:

Given $\mathit{scope}(C) = \{X, Y_1, \ldots, Y_n\}$, then C can be expressed as the subset S of $\langle D(X) \times D(Y_1) \times \cdots \times D(Y_n) \rangle$ which contains the assignments to $\mathit{scope}(C)$ which are allowed by C. The replacement of C is a new constraint over V_i and Y_i which allows the assignments $\{\langle v_1, \ldots, v_{|\overline{V}|}, y_1, \ldots, y_n \rangle | \ \langle f(\langle v_1, \ldots, v_m \rangle), y_1, \ldots, y_n \rangle \in S\}$.

Example 4.6. *Consider a set variable X of size 2 drawn from $\{0, 1, 2, 3\}$. This is represented under the explicit representation by $\langle \overline{V}, f \rangle$, where $\overline{V} = \langle V[1], V[2] \rangle$ and the $V[i]$ have domain $\{0, 1, 2, 3\}$, f is defined as mapping each pair of values $\langle V_1, V_2 \rangle$ to the set $\{V_1, V_2\}$ for those pairs where $V_1 \neq V_2$. Consider the constraint $\sum_{i \in X} = 3$, expressed as the set of tuples $\{\langle \{0, 3\} \rangle, \langle \{1, 2\} \rangle\}$. When this constraint is replaced by the representation $\langle \overline{V}, f \rangle$, then the original constraint is replaced by the constraint $\langle V[1], V[2] \rangle \in \{\langle 0, 3 \rangle, \langle 3, 0 \rangle, \langle 1, 2 \rangle, \langle 2, 1 \rangle\}$.*

An obvious requirement on mapping the constraints on a variable to the constraints on a representation of that variable is that the set of solutions must be preserved. Specifically, each solution of the original problem should be mapped to at least one solution of the new problem, and any solution of the new problem should denote a solution to the original problem. Theorem 4.4 show that this mapping is valid, and that the new CSP has a set of solutions for each original solution.

Theorem 4.4. *In this theorem, $\langle \overline{V}, f \rangle$ is a variable representation of a variable X with domain D. P is a CSP containing a variable X with domain D, an array of variables V and the constraint $c_R : f(\overline{V}) = X$.*

1. *Consider a CSP P containing X and the CSP P' generated by adding the variables $\overline{V}$ and the constraint $f(\overline{V}) = X$ to P. The solutions of P' are exactly generated by taking solutions to P and joining assignments to $\overline{V}$ which satisfy $f(\overline{V}) = X$, for the assignment to X in the solution.*

2. *P has the same solutions as the CSP P' generated by taking any constraint C in P (except c_R) where $X \in \mathit{scope}(C)$ and replacing X with $\overline{V}$ by $\langle \overline{V}, f \rangle$ in C.*

3. *If the only constraint in P which refers to X is the constraint $f(\overline{V}) = X$, a new CSP P'' can be generated by removing the variable X and the constraint $f(\overline{V}) = X$ from P. The solutions to P'' are exactly the solutions to P without the assignment to X.*

Proof.

1. For each assignment to X there exists at least one (and exactly one if the representation is perfect) assignment to $\overline{V}$ such that $f(\overline{V}) = X$ is satisfied.

2. Consider an assignment s to *scope*(P) which is a solution to P. This will be a solution to P' if and only if s satisfies C with X replaced by $\overline{V}$, as described in definition 4.8. However, s satisfies C and therefore in particular satisfies $f(\overline{V}) = X$. By the definition of replacing a variable in a constraint with a representation, the original constraint will be satisfied if and only if the new one is, as the constraint replaced is by replacing assignments to X with assignments to $\overline{V}$ which satisfy c_r. The reverse argument follows identically.

3. Clearly any solution to P when limited to *scope*(P'') will be a solution to P'', as the constraints on P' are a subset of the constraints on P. Given any solution to P'', adding the unique assignment to X which satisfies $f(\overline{V}) = X$ clearly generates a solution to P.

□

One minor problem with both Definition 4.8 and Theorem 4.4 is that they both assume that constraints are expressed extensionally. Most constraint solvers operate most efficiently on constraint expressed intensionally using a small language of constraints provided with the solver. How an intensional representation of a constraint can be refined to an intensional representation on a representation is research topic in its own right.

Fiona [45] performs this kind of mapping, and a more general system called Conjure [28] which the author of this thesis is involved with aims to map between representations and intensional representations of constraints while allowing arbitrary nesting of both constraints and constraint operators. As Conjure is not the topic of this thesis, where obvious the intensional version of refined constraints will simply be given, but a formal method as to how they can be generated will not be discussed further.

4.4 Partial Representations

One requirement of a representation is that every assignment to the variable must be represented. Similarly, in variable representations, each assignment to the representation must represent only a single assignment to the represented variable. There are interesting and useful representations which fail to satisfy this requirement, Example 4.7 gives one frequently occurring example, representing only the size of a set.

Example 4.7. *Given a variable X whose domains is all subsets of $\{1, 5, 10, 20\}$, the size of X can be represented by a single variable R of domain $\{0, 1, 2, 3, 4\}$, where the assignment i to R represents all sets of size i.*

The two constraints $|\mathbf{X}| < \mathbf{2}$ and $|\mathbf{X}| > \mathbf{3}$ map to the constraints $\mathbf{R} < \mathbf{2}$ and $\mathbf{R} > \mathbf{3}$, which have no solution.

In general however, R cannot be used as a representation of X. The constraint $\mathbf{5} \in \mathbf{X}$ maps to the constraint $\mathbf{R} > \mathbf{0}$, as for any other assignment to R, there is some set it represents which is allowed by $\mathbf{5} \in \mathbf{X}$. Similarly, the constraint $\mathbf{5} \notin \mathbf{X}$ maps to the constraint $\mathbf{R} < \mathbf{4}$. However while there is no solution to both $\mathbf{5} \in \mathbf{X}$ and $\mathbf{5} \notin \mathbf{X}$, there are a number of solutions to $\mathbf{R} > \mathbf{0}$ and $\mathbf{R} < \mathbf{4}$.

As the preceding example shows, partial representations are of limited use by themselves. When constraints do not line up with the partial representation, then it is possible to get invalid solutions. However, these partial representations still have a use in the context of combing representations and channelling, which will be discussed in the next section. Definition 4.9 gives the definition of partial representation, Definition 4.10 gives the definition of partial variable representation.

Definition 4.9. A **partial representation** $\langle R, f \rangle$ of a variable X is defined identically to Definition 4.4 on page 52, except it may not satisfy requirement 3.

Definition 4.10. A **partial variable representation** of a variable X is a pair $\langle \overline{V}, g \rangle$ defined as follows:

$\overline{V}$: An array of variables of length n

g: A function from the set of assignments of $\overline{V}$ to $\mathbb{P}(dom(X))$ such that all elements of $dom(X)$ are contained in the image under g of some assignment to $\overline{V}$.

The partial representation **induced** by a partial variable representation $\langle \overline{V}, g\rangle$ is the pair $\langle R, f\rangle$, defined as:

R**:** The lattice whose elements are the set $\{\langle s_1, s_2, \ldots, s_n\rangle | \forall i.\ s_i \subseteq dom(V[i])\}$ with the ordering $\forall\{r_1, r_2\} \subseteq R.\ (r_1 < r_2) \iff (\forall i.\ r_1[i] \subseteq r_2[i])$

f**:** The function defined over lists of sub-domains of $\overline{V}$, where $f(\overline{r})$ is the union of the sets $g(\overline{a})$ for every assignment $\overline{a}$ of $\overline{r}$.

4.5 Combining Representations

So far both good and bad representations have been demonstrated. Except the complete representation, every representation presented so far has had constraints on which it does not perform as well as the complete representation. This leads to the question of whether representations can be found.

One obvious way of trying to improve representations is to merge two existing representations into a single representation. This is closely related to channelling, a commonly used tool to improve CSP models. Joining representations or channelling should not be seen as a silver bullet, Section 7.2 shows how combining representations can in some cases drastically increase the size of search. When used with care, joining two representations can be used to improve representations.

Joining representations is closely related to channelling between different representations. In fact, channelling can be seen a method of implementing joining with weak propagation. Joining is stronger than channelling because with channelling, independent copies of problem constraints are placed on both the representations being channelled together. With joining a single constraint would be placed across both representations.

Joining a list of representations obviously will require joining their lattices. The general mathematical definition of joining a list of lattices is given in Definition 4.11, and some simple consequences are given in Lemma 4.5

Definition 4.11. Given a list of lattices $\langle R_1, \ldots, R_n\rangle$, the **join** of the lattices R_i is a new lattice $R = \langle S, \leq_R\rangle$ defined as follows:

- $S = \{\langle r_1, \ldots, r_n\rangle | r_i \in R_i\}$
- $\langle r_1, \ldots, r_n\rangle \leq_S \langle r'_1, \ldots, r'_n\rangle \iff (r_1 \leq r'_1 \wedge \cdots \wedge r_n \leq r'_n)$

Lemma 4.5. *Given a list of lattices $\langle R_1, \ldots, R_n \rangle$ and their join R_i as given in Definition 4.11, the following are true:*

- *R satisfies all the requirements of being a lattice.*
- $R^{\Uparrow} = \langle R_1^{\Uparrow}, \ldots, R_n^{\Uparrow} \rangle$
- $R^{\Downarrow} = \langle R_1^{\Downarrow}, \ldots, R_n^{\Downarrow} \rangle$

Proof. Obvious from definition of representation and join. □

Definition 4.12 describes how two representations are joined, by joining their lattices. This join is performed in the most straightforward and way, each state of the new representation is formed from a pair of states, one from each of the representations being joined.

Definition 4.12. The *join* of a list of n (possibly partial) representations $\langle R_i, f_i \rangle$ often denoted $\langle R_1, f_1 \rangle + \cdots + \langle R_n, f_n \rangle$ is $\langle R', f' \rangle$, where R' and f' are defined as follows:

- R' is the join of the lattices $\langle R_1, \ldots, R_n \rangle$.
- $f'(\langle r_1, \ldots, r_n \rangle) = f_1(r_1) \cap \cdots \cap f_n(r_n)$.

Theorem 4.6 shows that the join of a list of partial representations is another partial representation and if at least one of the representations is not partial, neither is the result.

Theorem 4.6. *The join of a list of partial representations is a partial representation. If at least one of the representations being joined is not partial, the result is not partial.*

Proof. To show the join $\langle R, f \rangle$ of a list of n partial representations $R_i = \langle M_i, g_i \rangle$ of a CSP variable X with domain D is a partial representation of X, it is sufficient to check three conditions.

1. f is a homomorphism, so for any pair of states r and r' of R, $r \leq r' \iff f(r_1) \subseteq f(r_2)$.
   As r and r' are elements of R, then $r = \langle m_1, \ldots, m_n \rangle$ and $r' = \langle m'_1, \ldots, m'_n \rangle$.
$$\begin{aligned} r \leq r' &\implies m_1 \leq m'_1 \wedge \cdots \wedge m_n \leq m'_n \\ &\implies (g_1(m_1) \subseteq g_1(m'_1)) \wedge \cdots \wedge (g_n(m_n) \subseteq g_n(m'_n)) \\ &\implies (g_1(m_1) \cap \cdots \cap g_n(m_n)) \subseteq (g_1(m'_1) \cap \cdots \cap g_n(m'_n)) \\ &\implies f(r_1) \subseteq f(r_2) \end{aligned}$$

2. The image under f of $R^{\Uparrow}$ must be D.
   By Lemma 4.12, $f(R^{\Uparrow}) = \bigcap_{i\in\{1,\ldots,n\}} g_i(M_i^{\Uparrow})$ and as the R_i are partial representations, then this is equal $\bigcap_{i\in\{1,\ldots,n\}} D = D$

3. The image under f of $R^{\Downarrow}$ must be $\emptyset$.
   Follows similarly to the proof for $R^{\Uparrow}$.

The only condition a partial representation must satisfy to be a representation is that if a state r allows assignment a, a is reachable from r. Without loss of generality assume R_1 is not partial. Then given any element $r = \langle m_1, \ldots, m_n\rangle$, if a is in $f(r)$ then a is in $g_1(m_1)$ so there exists m_1' in R_1 such that $g_1(m_1') = \{a\}$. Therefore the state $r' = \langle m_1', m_2, \ldots, m_n\rangle$ satisfies the condition that $f(r') = a$ and $r' \leq r$. □

The definition of the join of two representations has a very natural form when expressed for variable representations and this is given in Definition 4.13. The proof that the generated representation is the join of the original representations is a simple generalisation of Theorem 4.6.

Definition 4.13. The **join** of the variable representations $\langle\overline{V_1}, f_1\rangle, \ldots, \langle\overline{V_n}, f_n\rangle$ is the variable representation $\langle\overline{W}, g\rangle$ is defined as follows:

- $\overline{W}$ is the array $\langle V_1[1], \ldots, V_1[|\overline{V_1}|], V_2[1], \ldots, V_n[|\overline{V_n}|]\rangle$
- g is defined on assignments $\overline{w}$ of $\overline{W}$. Each assignment $\overline{w}$ can be split into an assignment $\overline{v_i}$ to each of the $\overline{V_i}$. If all the $f_i(\overline{v_i})$ take the same value for these assignments, $g(\overline{w})$ is defined and also takes this value. If any of the $f_i(\overline{v_i})$ are undefined or different, then $g(\overline{w})$ is undefined.

The most commonly join in this thesis is joining the *occurrence*, *explicit* or *Gent* representation with the partial representation given in Example 4.7. These will be denoted as the *occurrence + size*, *explicit + size* and *Gent + size* representations.

4.6 Representational Propagators

The traditional definition of propagators, as given by Definition 2.18 on page 28, does not immediately apply to representations. A generalisation to arbitrary representations is given in Definition 4.14. Note that propagators on representations are defined on the ordering of the elements of the representation, and not the subsets of the domain they represent.

Definition 4.14. Consider a constraint c over an array of variables $\overline{V}$, where $V[i]$ is represented by $\langle r_i, f_i \rangle$. A function P from the set $S = \{\langle r_1, \ldots, r_n \rangle | r_i \in R_i\}$ to itself is a **propagator** of c if it satisfies the following list of conditions:

1. Monotonic:
   $\forall \{\overline{s_1}, \overline{s_2}\} \subset S.\ \overline{s_1} \leq_{vec} \overline{s_2} \implies P(\overline{s_1}) \leq_{vec} P(\overline{s_2})$

2. Non-increasing:
   $\forall \overline{s} \in S.\ P(\overline{s}) \leq_{vec} \overline{s}$.

3. Preserves solutions:
   For all $\overline{s}$ in S, consider all arrays $\overline{a}$ in S such that each element of $\overline{a}$ represents a single assignment and $\overline{a} \leq_{vec} \overline{s}$. Then if the assignment represented by $\overline{a}$ satisfies c, then $\overline{a} \leq_{vec} P(\overline{v})$.

4. Identifies Assignments:
   For all $\overline{s}$ in S where $P(\overline{s})$ allows a single assignment to $\overline{V}$, then if c does not allow that assignment, $P(\overline{s}) = 0$.

The only part of Definition 4.14 which does not immediately follow from the previous definition of propagators is point 3. Rather than requiring all assignments which represent solutions to the constraint are preserved, a weaker possible definition is that for every solution, at least one assignment which represents it is preserved. The following example gives an example of this distinction and why it matters.

Example 4.8. *Consider a variable X with the domain of all subsets of size 2 of $\{1, 2, 3, 4, 5\}$, represented by the variable representation $\langle \langle y_1, y_2 \rangle, f \rangle$, where each of the y have domain $\{1, 2, 3, 4, 5\}$ and f is defined by $f(\langle y_1, y_2 \rangle) = \{y_1, y_2\}$, when $y_1 \neq y_2$.*

The constraint $\mathbf{1} \in \mathbf{X}$ is most naturally represented by $\mathbf{1} \in \mathbf{y_1} \vee \mathbf{1} \in \mathbf{y_2}$. However a stronger possible propagator is given by the following function $P(d_1, d_2)$, defined on a list of sub-domains where d_i is a sub-domain of y_i.

- *If $1 \in d_1$ set $d_1 = \{1\}$.*
- *If $1 \notin d_1$ and $1 \in d_2$, set $d_2 = \{1\}$.*

- *If $1 \notin d_1$ and $1 \notin d_2$, fail.*

This definition will clearly lose solutions to $\mathbf{1} \in \mathbf{y_1} \vee \mathbf{1} \in \mathbf{y_2}$, for example $y_1 \in \{1,2,3,4,5\}, y_2 \in \{1,2,3,4,5\}$ will be propagated to $y_1 \in \{1\}, y_2 \in \{2,3,4,5\}$, which has removed the assignment $\langle y_1, y_2 \rangle = \langle 2, 1 \rangle$. On the other hand, every assignment to X which was represented before is still represented.

It may appear that cases like Example 4.8 are weakening the "preserves solutions" part of the definition of propagator, requiring instead that only one image of each solution must be preserved, rather than all images of all solutions. This is however not the case, as Lemma 4.7 shows.

Lemma 4.7. *Replacing the 3rd condition in Definition 4.14 with:*

3 Preserves solutions:
For all $\overline{s}$ in S and all assignments $\overline{a}$ to $\langle f_1(s[1]), \ldots, f_n(s[n]) \rangle$, if $\overline{a}$ satisfies c, then there must be some $\overline{s'}$ in S such that $\overline{s'}$ represents only $\overline{a}$ and $\overline{s'} \leq P(\overline{s})$

leads to a logically identical definition.

Proof. Clearly this condition is weaker than the one in Definition 4.14, however there could be propagators which do satisfy this condition but do not satisfy the original Condition 3. Consider such a propagator P for a constraint c over a list of variables $\overline{V}$, where $V[i]$ is represented by $\langle r_i, f_i \rangle$, does exist.

If $S = \{\langle r_1, \ldots, r_n \rangle | r_i \in R_i\}$, then there must exist some $\overline{s}$ and $\overline{a}$ in S where $\overline{a}$ represents a single assignment to $\overline{V}$ which satisfies c, $\overline{a} \leq_{vec} \overline{s}$ but $\overline{a} \not\leq_{vec} P(\overline{s})$.

However P must be monotonic, so $P(\overline{a}) \leq_{vec} P(\overline{s})$. But each element of $\overline{a}$ allows only a single assignment and as c allows this list of assignments, $P(\overline{a}) = \overline{a}$, so $\overline{a} \subseteq P(\overline{a}) \subseteq P(\overline{s})$. □

The propagator in Example 4.8 is in fact invalid because it is not monotonic.Propagators are required to be monotonic is this ensures a well-defined fixed point regardless of the order in which propagators are executed. This makes studying the behaviour of such propagators much more difficult. The basic idea of trying to implement stronger propagators like in Example 4.8 still has merit and is discussed further in Section 8.5.

Definition 4.15 provides the most general and obvious method of mapping a traditional propagator, that is one on the complete representation, to a propagator where each variable has a representation. Lemma 4.8 proves that this definition always produces correct propagators on representations.

Definition 4.15. Given a propagator P of a constraint c on an array of variables $\overline{V}$ and an array of representations $\overline{R}$, where $R[i] = \langle r_i, f_i \rangle$ is a representation for $V[i]$, the **represented** propagator $P_{\overline{R}}$ of P, also called **P with respect to** $\overline{V}$ with respect to $\overline{R}$, is defined on representational states of $\overline{R}$ as:

$$\begin{aligned} P_{\overline{R}}(\overline{r}) = lub\{\overline{x} | \quad & \overline{x} \leq_{vec} \overline{r} \wedge \ \overline{x} \text{ allows one assignment to } \overline{V}, \\ & \text{which is allowed by } P(\ \langle f_1(r[1]), \ldots, f_n(r[n]) \rangle \ )\} \end{aligned}$$

Lemma 4.8. *Consider any propagator P of a constraint c on an array of variables $\overline{V}$ and an array of representations $\overline{R}$, where $R[i] = \langle r_i, f_i \rangle$. Then the represented propagator $P_{\overline{R}}$, as given in the preceding definition is a propagator as defined by Definition 4.14.*

Proof. In order to be a propagator, $P_{\overline{R}}$ must be monotonic, non-increasing, preserve solutions and identify assignments. These conditions will be checked in turn. Given the set $S = \{\langle r_1, \ldots, r_n \rangle | r_i \in R_i\}$:

1. Monotonic:
   Consider a pair $\overline{s_1}$ and $\overline{s_2}$ in S such that $\overline{s_1} \leq_{vec} \overline{s_2}$. For each $\overline{s_i}$, define $\overline{t_i} = \langle f_i(s_i[1]), \ldots, f_i(s_i[|\overline{V}|]) \rangle$. Then $\overline{s_1} \leq_{vec} \overline{s_2} \implies \overline{t_1} \leq_{vec} \overline{s_2} \implies P(\overline{t_1}) \leq_{vec} P(\overline{t_2})$.

   Looking at the definition of $P_{\overline{R}}$, $P_{\overline{R}}(\overline{s_1})$ is the lowest upper bound of the set:

   $$\begin{aligned} \{\overline{x} | \quad & \overline{x} \leq_{vec} \overline{s_1} \wedge \ \overline{x} \text{ allows one assignment to } \overline{V}, \\ & \text{which is allowed by } P(\ \langle f_1(s_1[1]), \ldots, f_n(s_1[n]) \rangle \ )\} \end{aligned}$$

   As $\overline{s_1} \leq_{vec} \overline{s_2}$, then $P_{\overline{R}}(\overline{s_2})$ is the lowest upper bound of a similar set, which contains at least all these elements. Therefore if $S \subseteq T \implies lub(S) \subseteq lub(T)$, the proof is done. This is true, because $lub(T)$ is greater than or equal to all the elements in T, and therefore all the elements of S so by definition $lub(S)$ must be less than or equal to $lub(T)$.

2. Non-increasing:
   Looking at the definition of $P_{\overline{R}}$, clearly $\overline{s}$ is greater than or equal to all the assignments it allows, so by definition $P_{\overline{R}}(s) \leq_{vec} \overline{s}$.

3. Preserves solutions and 4. Identifies solutions:
   Both true from definition.

□

4.7 Representational Symmetry

Rather than considering a CSP and the symmetry it contains, it can be enlightening to consider how that CSP was constructed and how the symmetry became introduced. In particular symmetry can be broadly split into two parts, those symmetries present in the original problem and those introduced during the modelling process, often by the use of representations.

A specification of the Social Golfers problem is given in Example 4.1 on page 49. The original problem does not have any obvious symmetry, yet the CP model given in Example 4.13 on page 72 has distinguished the golfers, the order of the weeks, the order of the games within each week and the order of the players within each game.

This split between problem and representation symmetry is not strictly defined for all problems. For example, it is unclear from the English specification if the Social Golfers problem should be modelled as a multiset of weeks, or as an ordered list of weeks (which would already have symmetry), however once a strict mathematical model of a problem has been found, the distinction between problem and modelling symmetry is clear.

The Social Golfers example demonstrates the major way in which symmetry is introduced, which is distinguishing a group of indistinguishable objects. This often occurs by refining a set or multiset into the *explicit* representation. By noting when symmetry is introduced in this way, it is possible to find some or all of the symmetries in a CSP model without having to search for them when the final CSP model is generated.

In general, a representation has symmetry when it is not simple (Definition 4.5 on page 53), so there is more than one assignment to the representation which represents the same assignment. Symmetry occurs in arbitrary representations, but defining how to deal with it in general is difficult due to a lack of structure. In the specific case of variable representations (Definition 4.6 on page 54), Definition 4.16 gives the definition of symmetry.

Definition 4.16. A variable representation $R = \langle \overline{V}, f \rangle$ has symmetry if there exists two distinct assignments $\overline{v_1}$ and $\overline{v_2}$ to $\overline{V}$ such that $f(\overline{v_1})$ and $f(\overline{v_2})$ are both defined and equal.

There are two different ways of viewing static symmetry breaking on variable representations. Either the representation itself can be changed to remove the symmetry, as in Definition 4.17, or a CSP the representation occurs in can have constraints added to it to break the symmetry, as in Definition 4.18. These two views are very similar and will lead to identical sets of solutions, but there will be differences in search space, as if the representation is changed so will any constraints on it.

Definition 4.17. Given two variable representations $R = \langle \overline{V}, f \rangle$ and $R' = \langle \overline{V}, f' \rangle$ of a domain D, R' is a valid symmetry breaking of R if for all assignments $\overline{v}$ to $\overline{V}$ then if $f'(\overline{v})$ is defined, then $f'(\overline{v})$ is equal to $f(\overline{v})$. R' is a complete symmetry breaking of R if R' also has no symmetry.

Given an ordering on the domain of each $V[i]$, the symmetry of R is lexicographically broken by the representation R' if R' is complete and for each assignment to D, the lexicographically least assignment to $\overline{V}$ which represents it is the one preserved.

Definition 4.18. Given a variable representation $R = \langle \overline{V}, f \rangle$ of a domain D, a constraint C on $\overline{V}$ is a valid symmetry breaking constraint for R if for all assignments d to D, there is at least one assignment $\overline{v}$ to $\overline{V}$ such that $f(\overline{v}) = d$ and $C(\overline{v})$ is true. C is complete if there is only one such assignment for each assignment to D.

Given an ordering on the domain of each $V[i]$, the symmetry of R is lexicographically broken by the constraint C if C is complete and for each assignment to D, the lexicographically least assignment to $\overline{V}$ which represents it is the one allowed by C.

While some minor issues with symmetry will arise in a number of places, the main study of symmetry in this thesis arises in Chapter 8.

4.8 Example Representations

Two closely related and very common types of variables in combinatorial problems are sets and multisets. Sets and multisets will be used both for running examples throughout this thesis, and also discussed specially in some sections. There are a number of representations of sets and multisets in common usage in constraint programming, and each of these will be defined and discussed in this section.

Note that none of the following definitions place any requirements on the elements of the sets and multisets being represented, although to ease presentation all examples will only consider sets and multisets of integers.

Probably the most common representation of a set or multiset drawn from a range of integers is the *occurrence* representation, given in Definition 4.19. In the case of sets, this representation is often known as the "characteristic function".

Definition 4.19. Consider a (multi)set variable X, whose domain is a set of multisets D, each drawn from a set S. Define *maxocc* to be $max_{s \in S, d \in D} occ(s, d)$. The **occurrence** variable representation of X is a pair $\langle \overline{V}, f \rangle$ where $\overline{V}$ and f are defined as follows:

- $\overline{V}$ is an array indexed by S of variables with domain $\{0, \dots, occ\}$.
- f maps assignments of $\overline{V}$ to the domain of X by $f(\overline{v}) = \{v[s] \times s | s \in S\}_m$, where the resulting (multi)set is in D.

$\overline{V}$ is referred to as the **occurrence array**.

One feature of Definition 4.19, which occurs in the other representations in this thesis, is that the representation does not have to represent all sub(multi)sets of a (multi)set S. For example, the domain of a set variable might contain only sets of one sizes. In this case, the representation function does not apply to assignments which represent a set of a different size.

Rather than change representations depending on the particular domain, an alternative is to impose constraints which restrict which (multi)sets are allowed. These two methods produce the same solutions, but are not the same. In particular, if the representation is restricted, propagators of constraints can then make use of this information, and in the case of GAC propagators are required to do so.

Example 4.9. *Consider a multiset variable X with domain $\{s \mid s \subseteq S, |s| \leq 4\}$, which is represented under the occurrence representation by an array $\overline{V}$. $X = \{1, 3, 3, 4\}_m$ is represented by the assignment $[1, 0, 2, 1]$ to $\overline{V}$. The assignment $[1, 2, 2, 0]$ does not represent any multiset, as it contains 5 values whereas the variable under consideration contains only sets with at most 4 values in them.*

The second most common set representation is the *explicit* representation, given in Definition 4.20. This representation is most commonly used for representing small sets of fixed size, often drawn from a very large set. The *ordered explicit* representation provides one method of breaking the symmetry of the explicit representation, making it simple.

Definition 4.20. Consider a (multi)set variable X, whose domain is a set of multisets D, each drawn from a set S and of the same size N. The **fixed-size explicit** variable representation of X is the array $\langle \overline{E}, f\rangle$ where $\overline{E}$ and f are defined as follows:

- $\overline{E}$ is an array of size N of variables with domain S.
- f maps assignments of $\overline{E}$ to the domain of X by $f(\overline{e}) = \{e[1], e[2], \ldots, e[Size]\}_m$, where this multiset is in D.

$\overline{E}$ is known as the **element array**.

The **ordered explicit** representation imposes the extra condition that under some ordering of S, the assignment to $\overline{E}$ must be ordered smallest to largest.

Example 4.10. *Consider X with domain $\{s \mid s \subseteq_m \{1,2,3,4,5\}, \forall i.\ occ(i,s) \leq c, |s| = 2\}$ represented by the explicit representation $\langle E, f\rangle$. Then E is an array of 4 variables, each of domain $\{1,2,3,4,5\}$. The assignments $\langle 1,1,4,5\rangle$ and $\langle 1,5,4,1\rangle$ to E both represent the assignment $\{1,1,4,5\}_m$ to X. Only the first of these would represent the assign in the ordered explicit representation. The assignment $\langle 1,1,1,2\rangle$ to E does not represent any assignment to X, as it has 3 occurrences of 1.*

There are two ways in which the *explicit* representation has been extended to deal with variable sized sets. The first, given in Definition 4.21, places a "dummy value" in the domain of each variable which is used to denote that a particular variable is not used. The second, given in Definition 4.22, is to have an extra array of Booleans which denotes which variables in the *element* array are in the (multi)set. An example of both of these is given after their definitions in Example 4.11

Definition 4.21. Consider a (multi)set variable X whose domain is a set of multisets D, each drawn from a set S. Define N to be $max_{d \in D}|d|$. The **explicit with dummy** variable representation of X is the pair $\langle \overline{E}, f\rangle$ where $\overline{E}$ and f are defined as follows:

- $\overline{E}$ is an array of length N of variables of domain $S \cup \{\emptyset\}$ (where $\emptyset$ represents a value not in S).
- f maps assignments of $\overline{E}$ to the domain of X by $f(\overline{e}) = \{e[i] | 1 \leq i \leq N \land e[i] \neq \emptyset\}_m$, where this multiset is in D.

$\overline{E}$ is known as the **element array**.

The **ordered explicit with dummy** representation, given an ordering of S, requires the assignment to $\overline{E}$ is ordered from smallest to largest, treating $\emptyset$ as if it were larger than everything in S.

Definition 4.22. Consider a (multi)set variable X whose domain is a set of multisets D, each drawn from a set S. Define N to be $max_{d \in D}|d|$. the **explicit with check** variable representation is the pair $\langle \overline{E}.\overline{C}, f \rangle$, where $\overline{E}$, $\overline{C}$ and f are defined as follows:

- $\overline{E}$ is an array of length N of variables of domain S.
- $\overline{C}$ is an array of length N of Boolean variables.
- f is defined by $f(\overline{e}.\overline{c}) = \{e[i] | 1 \leq i \leq N \land c[i] = \textsc{True}\}_m$, where this multiset is in D.

$\overline{E}$ is called the **element** array, and $\overline{C}$ is called the **check** array.

The **ordered explicit with check** representation requires that the elements of $\overline{C}$ which are TRUE come before all of those which are FALSE, and the elements of $\overline{E}$ are ordered from smallest to largest.

Example 4.11. *Consider a variable X with domain $\mathbb{P}(\{1,2,3,4,5\})$ represented by the explicit with dummy representation $\langle \overline{E}, f \rangle$ and by the explicit with check representation $\langle \overline{D}.\overline{C}, g \rangle$*

Then the assignments $\langle 1,2,3,\emptyset,\emptyset \rangle$ and $\langle 1,\emptyset,3,\emptyset,2 \rangle$ to $\overline{E}$ both represent the domain value $\{1,2,3\}$, as will any other permutation. Both of the assignments $\langle 1,2,3,4,5 \rangle$. $\langle T,T,T,F,F \rangle$ and $\langle 1,2,3,1,1 \rangle$. $\langle T,T,T,F,F \rangle$ to $\overline{D}.\overline{C}$ also represent $\{1,2,3\}$. Here as well as permuting the arrays, it is possible to assign any value to the element array when the check array is false.

These two methods of extending the *explicit* representation to variable-sized sets have their own advantages and disadvantages, mainly to do with symmetry, as demonstrated in Example 4.11. The major disadvantage of the *explicit with* *check* representation is that it introduces a large amount of extra symmetry, as when the i^{th} element of the *check* array is assigned FALSE, then the value of the i^{th} element of the *element* array is ignored. This symmetry cannot be captured by a permutation of the literals of the variables in the representation, so cannot be easily dealt with by existing symmetry breaking methods in CP. This means that while the *ordered explicit with dummy* representation is simple, the *ordered explicit with check* representation is not.

The major advantage of the element with check representation is that it is often easier to integrate with existing intensional constraints, which must often be rewritten to deal with the extra dummy value which is introduced by the *explicit with dummy* representation.

Lemma 4.9. *The explicit and explicit with check representations are not simple.*

Proof. It is possible to freely permute the array of variables in the *explicit* representation and they will represent the same (multi)set, so as long as this array has length greater than 1 and more than one possible assignment, the representation cannot be simple. $\square$

Given the common usage of the *occurrence* and *explicit* representations and the fact that (as will be shown later) they each have advantages and disadvantages, an obvious goal would be to attempt to construct a representation that combines their advantages. One naive way of doing this would be by joining them. The *Gent*[2] representation (Definition 4.23) is a new representation which combines some of the strengths of both the *occurrence* and *explicit* representations. This representation arose when trying to combine the practical benefits of both the *occurrence* and *explicit* representations. Later in this thesis it will be compared to the *occurrence* and *explicit* representations, which will demonstrate its theoretical advantages over these two representations.

Definition 4.23. Consider a (multi)set variable X whose domain is a set of multisets D, each drawn from a totally ordered set S. The **Gent** variable representation is a pair $\langle \overline{V}, f \rangle$, where $\overline{V}$ and f are defined as follows:

- $\overline{V}$ is an array of length $|S|$ of variables with domain $\{0, 1, \ldots, max(size)\}$.
- f maps an assignment $\overline{v}$ to $\overline{V}$ to an assignment of X using the following rules:

[2]Designed by the author and Ian Gent while modelling a particular problem and later generalised.

1. f is only defined if the **non-zero** elements of $\overline{V}$ are in strictly increasing order.
2. If $v[b] = 0$, $f(\overline{v})$ contains no occurrences of b.
3. If $v[b] \neq 0$, then given $z = max(\{v[i]|i < b\})$, $z = 0$ if b is the smallest element of S, then $f(\overline{v})$ contains $v[b] - z$ occurrences of b.
4. $f(\overline{v})$ is only defined if the previous rules define a multiset in the domain of X.

Example 4.12. *Given X with domain $\{s \mid s \subseteq_m \{1,2,3,4,5\}, |s| = 4\}$, consider X represented under the Gent representation by $\langle \overline{V}, f\rangle$ with the obvious ordering on $\{1,2,3,4,5\}$. Then $\langle 0,2,0,4,0\rangle$ represents $\{2,2,4,4\}_M$ and $\langle 1,2,4,0,0\rangle$ represents $\{1,2,3,3\}_M$. The representation of both $\langle 0,1,0,1,0\rangle$ and $\langle 0,2,0,1,0\rangle$ is undefined, as the non-zero elements are not in strictly increasing order.*

The final representation of sets which will be considered in this thesis is the lexicographic representation, given in Definition 4.25. Unlike the *occurrence*, *explicit* and *Gent* representations, the lexicographic representation cannot be considered as a variable representation.

Definition 4.24. Given a set S and an ordering on the elements of S, under the lexicographic ordering on subsets of S, $s_1 \leq_{lex} s_2$ if and only if the smallest element in s_1 is less than the smallest element in s_2, s_1 is empty or if the smallest element in both s_1 and s_2 is a, $s_1 - a \leq s_2 - a$.

Definition 4.25. Consider a (multi)set variable X whose domain is a set of multisets D, each drawn from a totally ordered set S. The **lexicographic** representation is defined by the pair $\langle R, f\rangle$, where R and f are defined as follows:

1. $R = \{\langle l, u\rangle | l, u \in D, l \leq_{lex} u\} \cup \{\emptyset\}$
2. $f(\langle l, u\rangle) = \{d | d \in D, l \leq_{lex} d \leq_{lex} u\}$ and $f(\emptyset) = \emptyset$.

The lexicographic ordering is of limited use by itself, but has been very successfully combined with the *occurrence* representation [71] to create a practically useful set and multiset representation. While the lexicographic representation can be usefully combined with other representations, it is hard to prove any results about it in isolation and therefore it will be only briefly discussed in this thesis.

4.8.1 Examples

Example 4.13 gives a model of the social golfers problem suitable for giving to a CP solver. This model is much more difficult to understand, as the high-level variables have been stripped away and the partitions replaced with a matrix of variables. This has massively increased the number of variables in the CSP and introduced symmetry.

Example 4.13. *The social golfers problem was defined originally in Example 4.1 on page 49. Here a more concrete model of the problem will be given, with the schedule represented as a 3 dimensional matrix.*

Here, as the partitions are being represented by matrices, it is necessary to impose two constraints that the matrix forms a correct partition, by imposing first that the players in each week are distinct, and secondly that no two players in two different games in the same week are the same.

Unlike in the previous model, the individual games in each week are now given a numerical label. The one dimensional matrix $Sched[i][j]$ *represents the golfers which play together in game* j *in week* i*.*

- *Given* n, p, w
- *Find Sched: 3D matrix indexed by* $[1..w][1..n/p][1..p]$ *with domain* $\{1..n\}$

  Such that

- $\forall i \in \{1, \ldots, w\}.\ \forall j \in \{1, \ldots, n/p\}.$ *AllDifferent(Sched[i][j])*
- $\forall i \in \{1, \ldots, w\}.\ \forall \{j, k\} \subseteq \{1..n/p\}.$
  $\forall a, b \in \{1..p\}.\ Sched[i][j][a] \neq Sched[i][k][b]$
- $\forall i \in \{1, \ldots, w\}.\ \forall j \in \{(i+1), \ldots, w\}.\ \forall l, k \in \{1, \ldots, n/p\}.$
  $(\sum_{x \in \{1,\ldots,p\}, y \in \{1,\ldots,p\}} .\ Sched[i,k,x] = Sched[j,l,y]) \leq 1$

4.9 Conclusion

This chapter has provided a framework for defining and working with representations, and shown how the common methods of implementing sets, multisets and other high-level types fit into this framework. The next two chapters will go on to show how representations in this framework can be usefully compared to each other and an informed choice made as to which should be used during search.

Chapter 5

Dominating Representations

Chapter 4 provided a framework for defining representations and using them to solve CSPs directly or to refine a CSP to another simpler CSP. The aim of this chapter is show how representations can be compared and how one representation can be proven to be "better" than another.

There are two major difficulties in comparing the performance of different representational choices in a CSP. The first is that for any specific CSP, the best possible representation will most likely be very closely related to the set of solutions. In the case of the Viewpoint formalisation (Subsection 3.1), this can lead to degenerate behaviour in the case where a problem has no solution. In general this will lead to a representation which is strongly affected by the list of solutions to the CSP.

This first difficulty is dealt with in this thesis by comparing representations over all problems constructed with a specific language of constraints. This has the obvious weakness that it will not be able to make use of problem-specific characteristics. However it has many advantages, including hopefully being easier to apply to a large range of problems and avoiding the problems of trying to find a better model for a single instance.

The second major problem is to choose which metric to use to compare representations. The obvious route would be to compare which choice of representation either finds a solution or proves no solution exists in the fastest possible time. This is unfortunately an unachievable goal at present and will probably continue to be unachievable. Solver-specific optimisations and implementation issues have a large effect on the time taken, with new releases of solvers and extensions for existing solvers often radically changing the speed taken to solve particular problems, for example when new propagators are added for new constraints. Therefore even if such a formalisation could be generated, it would have to be redesigned for each solver and corrected or restarted from scratch each time the solver was extended.

Instead, the metric used in this thesis is to compare the size of the search tree generated. A large number of factors still affect this metric, including the representations used, the variable and value orderings, the level of propagation used for each constraint and the symmetry breaking methods used. However, these are all possible to categorise exactly . So a solver-independent metric can be constructed and studied.

Clearly the results of this metric must be used with consideration. In particular, any change in representation which decreases search size at the cost of increasing the time taken per node of search will be better under this metric. In the most extreme case, if a CSP is modelled using only a single variable then propagation at the first node will either find all solutions or prove none exist. This representation will therefore be the best possible in terms of search size.

Given this proviso this thesis will show it is still possible to make powerful and useful comparisons of representations based on the search tree their usage in a problem generates. One particular reoccurring theme will be that in many cases it will be shown that substantially simplifying some part of search, for example replacing a representation of exponential size with one of polynomial size, will not increase the size of search. In this case using the simpler representation should hopefully always provide a quicker search.

Definition 5.1. Given two representations R_1 and R_2 of a domain D, R_1 **dominates** R_2 if given a CSP where R_2 is used to represent one of the variables, then for all static search orderings replacing R_2 with R_1 leads to a smaller or equal sized search.

R_1 **strictly dominates** R_2 if R_1 dominates R_2 and R_2 does not dominate R_1. R_1 **dynamically dominates** R_2 if R_1 also results in a smaller search tree for every possible dynamic search ordering as well.

Definition 5.1 gives the definition of one representation dominating another, in terms of search trees. One important result is that the definition of dynamic dominance given, which at first appears useful, is unfortunately unusable in the case of simple representations, as shown in Theorem 5.1. The proof of Theorem 5.1 shows that this is true as dynamic orderings provide too much freedom. This is because as soon as the search tree generated by different representations differs in any way, a dynamic heuristic then has free rein to move one or the other search tree directly to a solution, or away from a solution.

In a less forced way, any dynamic branching method can be ffected in unpredictable ways by the differences in representations, leading to searches which are almost impossible to compare. Therefore it is necessary to use only static variable orderings when considering theoretical performance.

Theorem 5.1. *Given two simple representations R_1 and R_2 of a domain D and a sub-domain s of X which is represented by R_1 and not R_2, R_1 will not dynamically dominate R_2 and R_2 will not dynamically dominate R_1.*

Proof. This basic idea of this proof is that a dynamic heuristic, having reached s, or the smallest sub-domain from R_2 which contains s can for one domain branch directly to a solution and in the other, create a large search tree.

Consider two CSPs P_1 and P_2, each containing two variables, X with domain D, and a Boolean variable Y. This CSP will have a single constraint, $Y = \textsc{True}$, which is propagated by the assignment propagator (that is, the propagator will wait until the Y is assigned, then fail if it has been assigned $\textsc{False}$). The only difference between the CSPs is that in P_i, X is represented using R_i.

Consider the first branch of search performing the propagation $X \in s$, and that this new constraint is GAC propagated.. If X is represented by R_1, then on this branch X will have sub-domain s. On the right branch it will have a different, larger sub-domain. At this point, the dynamic branching strategy could choose either to assign $Y = \textsc{True}$ and then any assignment to X, or assign $Y = \textsc{False}$ which would fail, then $Y = \textsc{True}$, then any assignment to X. A dynamic ordering could choose either to perform the smaller search if the current sub-domain of X is exactly s, or only if the sub-domain is larger than s. Therefore in different dynamic orderings, the search using R_1 or R_2 will be bigger. □

Definition 5.2 describes one representation embedding another. Representations dominating and embedding one another are closely related, and Theorem 5.3 shows that in the case of simple representations, the two are equivalent. For non-simple representations the relation is unfortunately more complex. Example 5.1 shows that increasing the number of representational states can actually lead to worse performance during search for non-simple representations.

Definition 5.2. $Rep_1 = \langle R_1, f_1 \rangle$ **embeds** $Rep_2 = \langle R_2, f_2 \rangle$ (or Rep_2 is **embedded in** Rep_1) if there is a function $M : R_2 \to R_1$ such that $\forall r \in R_2.\ f_1(M(r)) = f_2(r)$ and $\forall r_1, r_2 \in R_2.\ r_1 \leq r_2 \implies M(r_1) \leq M(r_2)$.

Rep_1 and Rep_2 are **equivalent** if Rep_1 embeds Rep_2 and Rep_2 embeds R_1. Rep_1 **strictly embeds** Rep_2 if Rep_1 embeds Rep_2 and Rep_2 does not embed Rep_1.

Example 5.1. *Given a set variable X with domain $\{s \mid s \subseteq \{1,2,3,4,5\}, |s| = 2\}$, consider representing it by the occurrence representation with occurrence array $\overline{V}$ and by the explicit representation with explicit array $\overline{W}$.*

For every sub-domain of $\overline{V}$ there is a sub-domain of $\overline{W}$ which represents the same assignments. This is generated by assigning as many values of $\overline{W}$ as are assigned in $\overline{V}$, then removing from the domains of the remaining variables any values which are disallowed by $\overline{V}$. Therefore the explicit representation embeds the occurrence representation.

Consider the CSP whose only variable is X with the constraints $1 \in X, 2 \in X$ and $3 \in X$. On the occurrence representation these become $V[1] = 1, V[2] = 1$ and $V[3] = 1$. Propagating these three constraints leads to the sum of X becoming greater than 2, so failure will occur.

On the explicit representation, these constraints become $W[1] = 1 \vee W[2] = 1, W[1] = 2 \vee W[2] = 2$ and $W[1] = 3 \vee W[2] = 3$. None of these constraints causes propagation beginning from complete domain, so search will have to be performed to show the CSP has no solution.

Dominance is very similar to the idea of expressitivity from [76], which simply compares representations by the set of sub-domains they represent. For simple representations, Lemma 5.2 shows that they are identical.

Lemma 5.2. *Given two simple representations $Rep_1 = \langle R_1, f_1 \rangle$ and $Rep_2 = \langle R_2, f_2 \rangle$ then the representation Rep_1 embeds the representation Rep_2 if and only if $\{f_2(r_2) | r_2 \in R_2\} \subseteq \{f_1(r_1) | r_1 \in R_1\}$.*

Proof. If Rep_1 embeds Rep_2, then there exists a function $M : R_2 \to R_1$ such that $\forall r \in R_2.\ f_1(M(r)) = f_2(r)$ and therefore $\forall r_2 \in R_2.\ \exists r_1 \in R_1.\ f_1(r_1) = f_2(r_2)$.

In the opposite direction, as the set of sub-domains represented Rep_2 is a subset of those represented by Rep_1 and as both representations are simple (and so each sub-domain is represented at most once), then the function M from R_2 to R_1 by $f(r_2) = r_1 \iff f_1(r_1) = f_2(r_2)$ demonstrates that Rep_1 embeds the representation Rep_2, as the ordering on simple representations is entirely determined by the domains they represent. □

Previous work on expressitivity in [76] showed that the *explicit* representation dominates the *occurrence* representation, as the *explicit* representation can represent a strict superset of the sub-domains represented by the *occurrence* representation. However this viewpoint is overly simplistic, as it assumes a representation is completely encapsulated by the list of sub-domains it represents and does not consider how it performs during serach. For non-simple representations it is necessary also to consider the relation between states, as well as the sub-domains those states represent. Considering only the states which can be represented can overestimate the effectiveness of a representation, as Example 5.2 demonstrates.

Example 5.2. *Consider representing two element multisets drawn from* $\{1, 2, 3\}$ *using the explicit representation both without symmetry breaking and with the symmetry breaking which orders the elements of the element array.*

The sub-domain $\{\{1, 1\}_M, \{1, 2\}_M, \{2, 3\}_M, \{1, 3\}_M\}$ *can be represented by the sub-domains* $\langle\{1, 2\}, \{1, 3\}\rangle$ *if symmetry breaking is not imposed, but if symmetry breaking is imposed the smallest sub-domains which allow this set of assignments is* $\langle\{1, 2\}, \{1, 2, 3\}\rangle$*, which also allows* $\{2, 2\}_M$*.*

Consider now representing the sub-domain $\{\{1, 2\}_M, \{1, 3\}_M\}$*. It can be represented exactly by the sub-domains* $\langle\{1\}, \{2, 3\}\rangle$ *both with and without symmetry breaking. However, these domains cannot be reached by removing values from the domains of* $\langle\{1, 2\}, \{1, 3\}\rangle$*.*

Without symmetry breaking, there are more possible sets of sub-domains which can be represented. This comes at the cost however that the domains which were shown second are not reachable from those which were generated first. It is therefore misleading to simply list the states which are reachable during search in the case of non-simple representations.

Theorem 5.3 shows the expected result that in the case of simple representations, dominance and embedding are equivalent. As Example 5.2 demonstrated, such a result will not apply to non-simple representations.

Theorem 5.3. *For simple representations* R_1 *and* R_2 *of a domain* D*,* R_1 *embeds* R_2 *if and only if* R_1 *dominates* R_2*. Furthermore if* R_1 *strictly embeds* R_2*,* R_1 *strictly dominates* R_2*.*

Proof. Consider two CSP implementations P_1 and P_2 which differ only that some variable X is represented by the representation R_i in P_i. Beginning from the complete representational states the weaker representation R_2 can only lead to the represented sub-domain of X in P_2 becoming larger than that of X in P_1. Similarly if the representational states in P_2 get larger, they can never get smaller than the ones in P_1 as P_2 can represent only a subset of the states allowed by P_1 and propagators are monotonic.

Now consider the case where R_1 dominates R_2. Given a sub-domain S of D and an element $s \in S$. A propagator $P_{s,S}$ for the constraint $X = s$ is defined as follows:

1. $\{s\} \subseteq d \subseteq S \implies P_{s,S}(d) = \{s\}$.
2. $\{s\} \nsubseteq d \subseteq S \implies P_{s,S}(d) = \emptyset$.
3. $d = \{x\}$ for some $x \neq s \implies P_{s,S}(d) = \emptyset$.
4. Otherwise, $P_{s,S}(d) = d$.

That $P_{s,S}$ is a correct propagator mostly follows trivially from the definition of propagator. The only non-trivial check is if $d_1 \subseteq d_2 \implies P_{s,S}(d_1) \subseteq P_{s,S}(d_2)$. If $d_2 \subseteq S$ then $d_1 \subseteq S$ and as $s \in d_2 \implies s \in d_1$, in this case the condition holds. If $d_2 \nsubseteq S_2$, then the conditional also holds.

Now, for each representation state r allowed by R_2, consider the sub-domain d represented by r. If $|d| = 1$, then R_1 must also have a representational state which represents d. Else, take an element $s \in d$ and consider the CSP implementation with a single variable X of domain D and the single represented $P_{s,d}$, where the first branch is $X \in d$, $X \notin d$, each propagated to GAC.

In the case of representing X by R_2, initial propagation will not reduce the domain. After imposing $X \in d$, the allowed sub-domain will be reduced to d, which will then be propagated to s, giving a solution. Therefore, if R_1 dominates R_2, it must also have a search of at most one node, and in order to do this R_1 must be able to represent the sub-domain d. Therefore R_1 embeds R_2.

In the case where R_1 strictly dominates R_2, R_1 must represent some sub-domain d of X which is not represented by R_2. Consider the CSP with two variables, one of them X (as required) and the other a Boolean variable Y represented by the complete representation and the three constraints $X \in d \implies Y = \textsc{True}, X \in d \implies Y = \textsc{False}, X \in d$, each of which is propagated to GAC. In P_1, propagating $X \in d$ will lead to X having exactly the sub-domain d, following which Y will be propagated to both FALSE and TRUE, which will empty its domain. In P_2 on the other hand, as X having the sub-domain d cannot be represented, after propagating $X \in d$, some values will be left in the domain of X which are not in d, and therefore the other two constraints will be unable to perform any propagation and search will have to continue. □

Theorem 5.3 and Lemma 5.2 together have shown that for simple variable representations, dominance and embedding are exactly equivalent. This means in this case, checking for dominance becomes much simpler.

5.1 Variable Representations

In general, showing one representation dominates another requires showing the mapping between them. With variable representations there is often a simpler way of showing one representation dominates another, by showing relations between the variables of the representations directly. Such relations are demonstrated in this thesis using channelling constraints, as given in Definition 5.3, with specific properties.

Definition 5.3. A set of **channelling constraints** between two variable representations $R_1 = \langle \overline{V_1}, f_1 \rangle$ and $R_2 = \langle \overline{V_2}, f_2 \rangle$ of a domain D is a set of constraints on the variables in $\overline{V_1}$ and $\overline{V_2}$ such that an assignment $\overline{v_1}$ to $\overline{V_1}$ and $\overline{v_2}$ to $\overline{V_2}$ satisfy the conjunction of the channelling constraints if and only if $f_1(\overline{v_1}) \equiv f_2(\overline{v_2})$.

Definition 5.4 shows three models of a permutation problem and the channelling constraints between them.

Definition 5.4. The three variable representations used in [44] to represent a permutation of the set $\{1, \ldots, n\}$ are:

Row Model: $\langle \overline{V_1}, f_1 \rangle$, where $\overline{V_1}$ is an array of length n of variables with domain $\{1, \ldots, n\}$, and f_1 maps an assignment to V_1 to the permutation where $V_1[i]$ is the i^{th} element of the permutation, where this is a valid permutation.

Column Model: $\langle \overline{V_2}, f_2 \rangle$, $\overline{V_2}$ is an array of length n of variables of domain $\{1, \ldots, n\}$, and f_2 maps an assignment to V_2 to the permutation where i is in the $V_2[i]^{th}$ position in the permutation, where this is a valid permutation

Boolean Model: $\langle \overline{V_3}, f_3 \rangle$, $\overline{V_2}$ is a 2 dimensional matrix of Booleans indexed $\{1, \ldots, n\}$, where f_3 maps V_3 to the permutation where i is in the j^{th} position if $V_3[i][j]$ is true, where this is a valid permutation.

The channelling constraints between these models are $V_1[i] = j \iff V_2[j] = i \iff V_3[i][j] = 1$

Unfortunately there does not appear to be a compact and simple way of checking if one variable representation dominates another. In practice, both in this thesis and in any paper referenced in this thesis there are three patterns which can be used to explain where one variable representation dominates another and one pattern which is used to generate equivalent representations. These three patterns are defined in the next three subsections.

5.1.1 Extra information by adding variables

The easiest kind of extension is adding extra variables, this idea is outlined in Definition 5.5. This is identical to joining a representation with a partial variable representation, as given in Definition 4.12 on page 60. The reason an alternative definition is given here is to unify the different methods of extending representations. It is obvious that adding extra variables will create a representation which dominates the original (although not necessarily strictly).

Definition 5.5. Given 2 variable representations $R = \langle \overline{V}, f \rangle$ and $R' = \langle \overline{V'}, f' \rangle$, R' is R augmented with extra variables if:

1. The first $|\overline{V}|$ elements of $\overline{V'}$ are exactly the same as $\overline{V}$.
2. Given any assignment $\overline{v'}$ to $\overline{V'}$, take the assignment $\overline{v}$ to $\overline{V}$ by taking the first $|\overline{v}|$ values from $\overline{v'}$. Then either $f(\overline{v}) \equiv f'(\overline{v'})$ or $f'(\overline{v'})$ is undefined.
3. Given any assignment $\overline{v}$ to $\overline{V}$, there exists at least one assignment $\overline{v'}$ to $\overline{V'}$ where $f(\overline{v}) \equiv f'(\overline{v'})$ and the first $|\overline{v}|$ elements of $\overline{v'}$ are equal to $\overline{v}$.

5.1.2 Extra information by splitting literals

Another method of creating a new representation from an old one in such a way it always dominates the original is to split a single literal into multiple literals, while keeping all other parts of the representation the same.

Example 5.3. *Consider representing an integer between 0 and 15 in binary by a Boolean array of length 4. The last variable in the array represents if the variable is even or odd. This could be extended to 3 values, to represent one of even, odd non-prime or odd prime.*

This allows for more representational information, as by setting this to either odd non-prime or odd prime at the start of search allows representing a sub-domain which could not be represented before.

Splitting literals is defined in Definition 5.6, and a proof that it creates dominating representations is given in Lemma 5.4. This Lemma does not give a proof of when the new representation strictly dominates, as proving when this occurs is a very difficult condition which would be hard to check.

Definition 5.6. Given 2 variable representations $R = \langle \overline{V}, f \rangle$ and $R' = \langle \overline{V'}, f' \rangle$, the domain element e of $V[i]$ is split into a set s in V' if:

1. For all $j \neq i$. $dom(V[j]) = dom(V'[j])$
2. Given $d = dom(V[i]) \cap dom(V'[i])$ then $e \notin d, d \cap s = \emptyset$, $dom(V[i]) = d \cup \{e\}$ and $dom(V'[i]) = d \cup s$
3. Given any assignment $\overline{v}$ to $\overline{V}$ which does not assign e to $V[i]$, $f(v) = f'(v)$.
4. Given an assignment $\overline{v'}$ to $\overline{V'}$ which assigns an element of s to $V'[i]$, then the assignment $\overline{v}$ which is identical except it assigns e to $V[i]$ satisfies $f'(\overline{v'}) \equiv f(\overline{v})$.

Lemma 5.4. *Consider two representations $R = \langle \overline{V}, f \rangle$ and $R' = \langle \overline{V'}, f' \rangle$ where the domain element e of $V[i]$ is split into a set s in R'. Then R' dominates R.*

Proof. To show R' dominates R, it is sufficient to show for each sub-domain of $\overline{V}$ there is a sub-domain of $\overline{V'}$ which represents the same set of assignments and these sub-domains of $\overline{V'}$ have the same ordering as the subsets of $\overline{V}$ had. This is very simple to see, by taking any subset of $\overline{V}$ and if that sub-domain has e in the sub-domain of $V[i]$, then replacing it with s. The result then follows directly from the definition of literal splitting. □

5.1.3 Extra information by variable/value swapping

The variable/value swap is a generalisation of the mapping given in the first two cases of Definition 5.4. In that case it is required that every value is taken by exactly one variable. In the more general case which is descibed in Definition 5.7, the weaker requirement that each value must be assigned to at most one variable is used. This requires adding to each variable in the swap a domain value which represents that a particular value is assigned to no variable. Example 5.4 gives an example of this.

Definition 5.7. Consider a variable representation $R = \langle \overline{V}, f \rangle$, where each variable in $\overline{V}$ has domain of size n and whenever $f(\overline{v})$ is defined, the elements of $\overline{v}$ are all different.

In this case, the *variable/value swap* of R can be defined, which is the new representation $R' = \langle \overline{V'}, f' \rangle$ where $\overline{V'}$ is an array of length n with domain size $|\overline{V}| + 1$. The purpose of this mapping is that $V'[i] = j$ will represent $v[j] = i$. The extra value in the domain of the variables in $\overline{V'}$ is to allow the possibility that some assignments to the V[i] never occur.

More precisely, the function $Map : \overline{V'} \rightarrow \overline{V}$ is defined only on those assignments to $\overline{V'}$ where all values from 1 to $|\overline{V}|$ occurs exactly once. In this case an assignment to $\overline{V'}$ is generated by $v[i] = j \iff v'[j] = i$. This will give exactly one assignment to each $v[i]$. Then f' is defined as $f'(\overline{v'}) = f(Map(\overline{v'}))$.

Example 5.4. *Consider the explicit representation $R = \langle \overline{V}, f \rangle$ of a fixed sized subset, size 3, of $\{1, 2, 3, 4, 5\}$. The variable/value swap of this will be given by a representation $R' = \langle \overline{V'}, f' \rangle$. If the assignment to $V'[i] = 4$, that means that no variable in $\overline{V}$ is assigned i, else the value of $V'[i]$ tells us which of the elements of $\overline{V}$ is assigned i. Some examples of mapping assignments to $\overline{V'}$ to $\overline{V}$ are:*

$Map(\langle 1,4,2,4,3 \rangle) = \langle 1,3,5 \rangle$, $Map(\langle 1,2,3,4,4 \rangle) = \langle 1,2,3 \rangle$,
$Map(\langle 3,4,2,4,1 \rangle) = \langle 5,3,1 \rangle$, $Map(\langle 3,4,4,1,2 \rangle) = \langle 4,5,1 \rangle$,
$Map(\langle 1,4,4,3,4 \rangle)$ *: Undefined - No assignment to 2nd variable.*
$Map(\langle 1,4,3,1,2 \rangle)$ *: Undefined - Two assignments to 1st variable.*

Lemma 5.5 proves that the simple variable/value swap given in the first two parts of Definition 5.4 produces two equivalent representations, as each dominates the other. However, in the general case, performing a variable/value swap produces a representation which strictly dominates the original representation.

Lemma 5.5. *Consider two variable representations $R_1 = \langle \overline{V}, f \rangle$ and $R_2 = \langle \overline{W}, g \rangle$ of a domain D where R_2 is the variable/value swap of R_1. Then R_2 dominates R_1.*

Proof. Without loss of generality, assume the domain of each member of $\overline{V}$ is $\{1, 2, \ldots, n\}$ for some n and the domain of each variable in $\overline{W}$ is $\{0, 1, \ldots, |\overline{V}|\}$, where $W[i] = 0$ denotes no element of $\overline{V}$ is assigned i. Showing R_2 dominates R_1 requires showing for each sub-domain of $\overline{V}$ there exists a sub-domain of $\overline{W}$ which represents the same sub-domain of D.

Given a sub-domain $\overline{v}$ of $\overline{V}$, generate a sub-domain $\overline{w}$ of $\overline{W}$ using the following rule: $dom(w[j]) = \{i | j \in dom(v[i])\} \cup \{0\}$. This sub-domain satisfies the required property. $\square$

5.1.4 Booleanize a variable

One common trick used in constraint programming, particularly when mapping a constraint program to SAT, is mapping a single variable X of domain $\{1, \ldots, d\}$ to an array $\overline{V}$ of Boolean variables of length d where $X = i$ if and only if $V[i]$ is True. One obvious example of this is turning the row model of a permutation into a column model of a permutation, comparing models 1 and 3 in Definition 5.4 on page 79.

Booleanizing a variable is defined in Definition 5.8, and a proof that it creates an equivalent representation is given in Lemma 5.6.

Definition 5.8. Given 2 variable representations $R = \langle \overline{V}.\langle x \rangle, f \rangle$ and $R' = \langle \overline{V}.\overline{W}, f' \rangle$, x is Booleanized by $\overline{W}$ in R' if:

1. $\overline{W}$ is an array of Boolean variables indexed by $dom(x)$.
2. f' is undefined unless exactly one element of $\overline{W}$ is True.
3. Given an assignment $\overline{v}$ to $\overline{V}$ and $\overline{w}$ to $\overline{W}$ where the only element of $\overline{w}$ which is True is $w[i]$, then $f'(\overline{v}.\overline{w})$ is equal to $f(\overline{v}.\langle i \rangle)$, or undefined if $f(\overline{v}.\langle i \rangle)$ is undefined.

Lemma 5.6. *Consider two representations $R = \langle \overline{V}.\langle x \rangle, f \rangle$ and $R' = \langle \overline{V}.\overline{W}, f' \rangle$ where x is Booleanized by $\overline{W}$. Then R' and R are equivalent.*

Proof. To show R' and R are equivalent, a mapping between sub-domains of $\overline{V}.\langle x \rangle$ to $\overline{V}.\overline{W}$ is required. As $\overline{V}$ is the same in both, only a mapping from x to $\overline{W}$ is required.

Given a sub-domain of x, map this to $\overline{W}$ by adding $W[i] = 0$ for all variables and $W[i] = 1$ for those values i in the sub-domain of x.

Given a sub-domain of $\overline{W}$, for it to represent a valid assignment there must be at least one value whose sub-domain contains 1, and no more than one value whose sub-domain does not contain 0. If the sub-domain of $W[i]$ is $\{1\}$, then generate the domain of x containing only 1. Otherwise, generate the sub-domain containing every i where the sub-domain of $W[i]$ contains 1.

These two mapping demonstrate the two representations are equivalent. □

5.2 Set Representations

In this section, as a concrete example of dominance, there is a number of dominance results shown for the three set representations (*occurrence*, *explicit* and *Gent*) and some common variations of them. In particular, this shows how the *Gent* representations dominates both the *occurrence* and *explicit* representations for representing sets.

Lemma 5.7 and Corollary 5.8 show the Gent representation embeds the *occurrence* representation and the *Gent + Size* representation embeds the *occurrence + size* representation. As the Gent representation is simple, this means it also dominates the *occurrence* representation.

Lemma 5.7. *The ordered Gent representation embeds the occurrence representation when representing sets of fixed and variable size.*

Proof. Expand all the '1' literals of the *occurrence* rep to get the *Gent* representation □

Corollary 5.8. *The Gent + size representation embeds the occurrence + size representation for representing sets.*

Proof.

This also follows from Lemma 5.7, noting that the size variable in both representations will be assigned the same sub-domain. □

Unfortunately, the Gent representation does not dominate the occurrence representation when representing multisets, as presented in Example 5.5.

Example 5.5. *Given a variable with domain* $\{s \mid s \subseteq_m \{1,2,3\}, \forall i.\ occ(i,s) \leq 2\}$*, consider representing this in the occurrence representation with occurrence array* $\overline{V}$ *and in the Gent representation with Gent array* $\overline{G}$*.*

Consider the sub-domain $\langle\{0,1,2\},\{0,2\},\{0,1,2\}\rangle$ *of* $\overline{V}$*, which represents all multisets that do not contain one occurrence of* 2*. Consider representing this as a sub-domain of the Gent representation. The assignment* $\langle 0,2,3\rangle$ *to* $\overline{G}$ *represents* $\{2,2,3\}_m$ *and the assignment* $\langle 1,3,0\rangle$ *represents* $\{1,2,2\}_m$*. Therefore the assignments used in these assignments must be contained in any sub-domain of* $\overline{G}$ *which* *allows all multisets which do not contain one occurrence of* 2*. However, allowing these assignments allows* $\langle 1, 2, 3\rangle$*, which represents the multiset* $\{1, 2, 3\}_m$*, which contains exactly one occurrence of* 2*. Therefore the Gent representation does not dominate the occurrence representation for multisets.*

Lemma 5.9 shows that the *Gent* representation also dominates the *ordered explicit* representation.

Lemma 5.9. *The Gent representation embeds the ordered explicit representation for both fixed and variable sized sets.*

Proof. Consider representing a set variable S of maximum size n with the Gent representation using an array of variables $\overline{G}$ and with the *explicit with dummy* representation using the array of variables $\overline{E}$. Then $\overline{G}$ is a variable/value swap of $\overline{E}$, with 0 in the domain of the variables in $\overline{G}$ denoting that a particular value is taken by no value in $\overline{E}$. □

5.3 Equivalence

Definition 5.2 on page 75 states that given two representations, if A dominates B and B dominates A, then A and B are equivalent. The work so far in this chapter suggests that given a set of equivalent representations, there should be no reason to choose one over any of the others. One particularly prominent class of equivalent variable representations are literal equivalent representations. These are created by rearranging the set of literals, but having the same underlying set. These are defined in Definition 5.9 and proved equivalent in Lemma 5.10.

Definition 5.9. Two representations $R_1 = \langle \overline{V_1}, f_1\rangle$ and $R_2 = \langle \overline{V_2}, f_2\rangle$ are literal equivalent if there exists a valid set of channelling constraints between $\overline{V_1}$ and $\overline{V_2}$ each of the type $V_1[i] = j \iff V_2[k] = l$ where every literal in both $\overline{V_1}$ and $\overline{V_2}$ appears in exactly one constraint.

Lemma 5.10. *Two literal equivalent representations are equivalent, that is they each dominates the other.*

Proof. Consider two literal equivalent representations $R_1 = \langle \overline{V_1}, f_1\rangle$ and $R_2 = \langle \overline{V_2}, f_2\rangle$ of a variable X. The channelling constraints between the representations generate a unique bijection between the literals of $\overline{V_1}$ and $\overline{V_2}$. Therefore given any sub-domain of $\overline{V_1}$, a unique sub-domain of $\overline{V_2}$ can be generated by applying this bijection to each literal in the sub-domain of $\overline{V_1}$. For any assignment $\overline{v_1}$ to $\overline{V_1}$ such that $f(\overline{v_1})$ is defined, there must be a unique assignment $\overline{v_2}$ to $\overline{V_2}$ such that these two assignments together satisfy the channelling constraints.

This means that the sub-domain of $\overline{V_2}$ generated by taking a sub-domain of $\overline{V_1}$ and mapping each literal to $\overline{V_2}$ must represent the same set of assignments. This mapping shows R_1 dominates R_2, and by symmetry R_2 also dominates R_1. □

There have been a number of previous papers however which have shown that using multiple literal-equivalent representations can provide a large reduction in search size. For example, Hnich et al. [44] compare the representations of permutations, given in Definition 5.4, which Lemma 5.11 shows are equivalent.

Lemma 5.11. *The 3 variable representations of a permutation of the integers* $\{1, \ldots, n\}$ *given in Definition 5.4 on page 79 are equivalent.*

Proof. Using the channelling constraints $V_1[i] = j \iff V_2[j] = i$ show the row and column models are equivalent, and Booleanization shows that the Boolean model is equivalent to both the row and column models. Therefore the result follows. □

Looking at [44], part of the difference between representations arises because the constraint that the variables form a permutation is imposed in different ways. However, this does not appear to explain all the differences, and some fundamental ones appear to remain, in particular the best model is to use either a variable for each row or each column, and not the binary model.

There are three reasons why the work on viewpoints shows that multiple representations can lead to smaller search space. The most simple reason is that there is an implicit constraint on all variables which impose exactly one literal in the variable's domain will be in any solution. Given two literal equivalent representations, these implicit constraints can change between models.

This effect can be easily seen on permutation problems. The row model (Definition 5.4) of a permutation does not require a constraint that each value is taken exactly once, the column model does not require a constraint that each variable takes exactly one value. The Boolean model must impose both of these constraints explicitly.

For a correct representation these constraints should be imposed, as they form part of the requirement that an assignment to the variables is a valid assignment to the representation, but often they are not imposed. Clearly it is possible to explicitly impose these constraints.

The second reason, shown in the work on viewpoints in Section 3.1, is that the most natural method of imposing constraints on different models may lead to different levels of propagation. In particular, if a solver requires all constraints are binary then different models may have very different sets of constraints. Also, an intensional constraint which a solver can efficiently represent on one model may have to be imposed as a table constraint on another model.

However, as presented in Section 3.8 of [70], propagators are implemented by querying and removing values from the set of literals in variables in the constraint's scope. Therefore, given any propagator and a literal equivalent representation, it is possible to adapt the algorithm to work directly on the literal equivalent representation and achieve the same level of propagation there.

The third reason shown in the literature why equivalent representations can be useful in search is that they may both guide search heuristics and allow different methods of branching. While it is true that when using a specific existing heuristic, for example choosing the variable with the smallest domain, it is possible to simulate running a heuristic on each of the literal equivalent representations and seeing which branching choice would have been made and mapping that to any of the other representations.

The one major drawback to trying to map branching constraints from one representation to a literal equivalent one, as given in Example 5.6, is that a set of branching constraints mapped to an equivalent model may appear to be invalid, as they forbid assignments. However, it is obvious that these assignments cannot represent solutions.

Example 5.6. *Consider a permutation of $\{1, \ldots, n\}$ represented by two variable representations $\langle \overline{V}, f\rangle$ and $\langle \overline{W}, g\rangle$. Both $\overline{V}$ and $\overline{W}$ are arrays of n variables of domain $\{1, \ldots, n\}$. $f(\overline{v}) = \langle v[1], \ldots, v[n]\rangle$, where this forms a permutation. $g(\overline{w})$ is the permutation where i appears in the $w[i]^{th}$ position, where this forms a permutation. These two models are connected by the set of channelling constraints $\{V[i] = j \iff W[j] = i | i, j \in \{1, \ldots, n\}\}$.*

On the first of these models, one valid set of branching constraints is all assignments to the first variable, generating the constraints $\{V[1] = 1, V[1] = 2, \ldots, V[1] = n\}$. On the second model, this is equivalent to the set of constraints $\{W[1] = 1, W[2] = 1, \ldots, W[n] = 1\}$. None of these branching constraints allows the assignment $\overline{L} = \langle 1, 1, \ldots, 1\rangle$. However in practice this is not a problem, as *this assignment does not represent a valid assignment to the representation.*

5.4 N-way Limited Representation

While a number of dominance results between different representations have now been shown, so far no result in this thesis provides a method of comparing the *explicit* and *occurrence* representations. The *explicit* representation, being non-simple, is difficult to effectively study and categorise. In particular, as discussed earlier, just looking at the set of sub-domains that can be represented can be insufficient to properly study non-simple representations. In this section a study of the *explicit* representation and some interesting properties which often arise during search and make comparison simpler will be discussed.

Definition 5.10. Given an *explicit* representation of a domain D with *explicit* array $\overline{E}$ and check array $\overline{C}$, a sub-domain of $\overline{E}$ and $\overline{C}$ is *n-way limited* if all variables in $\overline{E}$ which do not have a single value left in the sub-domain have the same sub-domain.

For reasons of symmetry, which will be explained in this section, the representational state of the *explicit* representation are often n-way limited, given in Definition 5.10, during search. There are two ways in which the sub-domain of a variable can change during search, either through propagation of a problem constraint, or by propagation by a branching constraint. Lemma 5.12 shows that propagation of problem constraints must map n-way limited sub-domains to other n-way limited sub-domains.

Lemma 5.12. *Given an explicit representation with explicit array E of the domain D of a variable X, then given any propagator for a constraint which involves X, it is possible to create a stronger propagator which always maps n-way limited states to n-way limited states which takes at most $O(|\overline{E}|.|dom(E[1])|)$ time.*

Proof. Take some n-way limited set of sub-domains of $\overline{E}$, called $\overline{D}$. Without loss of generality, assume that there exists i such that forall $j \leq i$, $|D[j]| = 1$ and for all $j \geq i$, all the $D[j]$ have an identical sub-domain. Now apply the propagator and assume the constraint does not fail. This may have reduced the domains of some of the $D[j]$ where $j > i$.

For any z which was removed from the domain $D[j]$, but not $D[k]$, for $j, k > i$, then we can remove z from the domain $D[k]$. If this was not the case there would have to be an assignment to E where $E[k] = D$. In this case, consider the same assignment with the j^{th} and k^{th} position swapped. This assignment clearly represents the same multiset, and was permitted before propagation, as $D[j]$ and $D[k]$ were the same. Therefore propagation should not have removed it and was invalid.

Therefore, after any propagation, it is permitted to look at al variables which had been unassigned before, take the union of all values removed from all sub-domains, and removed these from all the sub-domains, leaving the resulting sub-domains n-way limited. □

Performing n-way branching will obviously map n-way limited sub-domains to other n-way limited sub-domains. Given this restriction on branching, Lemma 5.13 shows that the *occurrence* representation and n-way limited representations can represent exactly the same set of sub-domains.

Theorem 5.13. *The occurrence representation and the explicit representation without symmetry breaking and allowing only "n-way limited" sub-domains are equivalent when representing sets.*

Proof. Consider a CSP variable X with domain $\mathbb{P}(S)$ for a fixed set S represented under the *occurrence* representation by *occurrence* array $\overline{O}$ and *explicit* representation by *explicit* array $\overline{E}$. Given a sub-domain of $\overline{O}$, construct an n-way limited sub-domain of $\overline{E}$ as follows:

1) For each element $O[x]$ of $\overline{O}$, if $O[x]$'s sub-domain is $\{1\}$, choose a variable of $\overline{E}$ and assign it x.

2) Allow each other variable to take all values except those x for which the sub-domain of $O[x] = \{0\}$.

Given an n-way limited sub-domain of $\overline{E}$, we can construct a sub-domain of $\overline{O}$ by a similar process:

1) The set S denotes the values in those elements of $\overline{E}$ whose sub-domain contains more than one values. as $\overline{E}$ is n-way limited, these domains must all contain the same values.

2) Define the sub-domain of $O[x]$ as follows: If some sub-domain is assigned x, then $\{1\}$. If no sub-domain contains x, then $\{0\}$. If no variable is assigned x, but $x \in S$, then $\{0, 1\}$.

It is easy to check that this sub-domain of $\overline{O}$ represents the same sets as $\overline{E}$. □

Theorem 5.13 shows that as long as n-way branching is performed during search, a set represented by the *explicit* representation will remain n-way limited throughout all of search.

However, there are still differences between the n-way *explicit* representation and *occurrence* representation which mean the *occurrence* representation is strictly better during search. In particular, while the *occurrence* representation is simple, the n-way *explicit* representation still is not. Therefore the n-way *explicit* representation cannot perform better than the *occurrence* representation, but can still perform worse.

This leads to the obvious question of whether the *explicit* representation should ever be used, as in this situation there is no reason to use the *explicit* representation, except for space reasons. This result is limited however, as in general symmetry breaking will be applied to an *explicit* representation.

Simply applying symmetry breaking however is of limited effectiveness as while it removes extra values from the domains of variables, this may not improve the performance. Examining the particular constraints which are being used can show if they are making use of the extra benefits from values removed by symmetry breaking. This will be discussed in Section 8.4.

Even with symmetry breaking however, the *explicit* representation is still unable in general to make use of the fact some value has been deduced to be in the set. Therefore the *occurrence* representation is preferable in situations where during search it will be deduced many values are in the (multi)set. The *explicit* representation is better when there are constraints relating different members of the sets, and propagation is more frequent either removing values from the set or assigning values at the current upper and lower bounds of the set.

5.4.1 Monotonic (Multi)set Constraints

Many commonly occurring constraints on sets and multisets are *monotonic* (Definition 5.11), and this property can have important consequences on the choice of representation. Example 5.7 gives an example of a monotonic constraint and a problem whose constraints are monotonic.

Definition 5.11. A constraint C is **monotonically set decreasing** with respect to a set $S \in \mathit{scope}(C)$ if given any assignment a to $\mathit{scope}(C)$ which satisfies C, then replacing the assignment s to S in a with any set t such that $t \subseteq s$ leads to a new assignment which also satisfies C. **monotonically set increasing** is defined in the obvious manner.

Example 5.7. *Given a set variable S and a fixed set t, the constraint $t \not\subseteq S$ is monotonically set decreasing. The constraints of the Golomb ruler problem (prob 6 at www.csplib.org) can be specified in this form:*

$$\forall \{i,j\},\{k,l\} \subseteq \{1,\ldots,n\}.\ \{i,j\} \neq \{k,l\} \implies \{i,j,k,l\} \not\subseteq S$$

As has been shown previously, *explicit* representations can easily represent propagation when it can be deduced that some value cannot be present in a (multi)set variable. However the *explicit* representation performs poorly when some value is deduced to be present in a (multi)set variable. This means it may be expected that the *explicit* representation will perform well on *monotonically set decreasing* constraints and badly on *monotonically set increasing* constraints. The first of these is exactly what theorem 5.14 proves.

Theorem 5.14. *Given a problem consisting of only monotonically decreasing constraints on some set variable S, then representing S by the explicit representation with dynamic symmetry breaking or occurrence representations will produce identical search trees. (assuming n-way branching).*

Proof. In theorem 5.13 it was shown that the *n-way limited explicit* representation and *occurrence* representation are equivalent in terms of representation power. Assigning a value in the *occurrence* representation 0 will propagate to the *n-way limited explicit* representation as normal, but if a variable is assigned 1 by propagation, then this is not propagated by normal propagation algorithms.

Given a *monotonically decreasing* constraint however, this kind of propagation can never occur. □

Note that theorem 5.14 only refers to representations of sets, not multisets. With multisets the weakness of *explicit* representations returns. Although *explicit* representations can represent that **no** occurrence of some value is contained in a (multi)set variable, it is much more difficult to represent that a limited number can occur, as this requires removing the value from some, but not all, of the element variables in the *explicit* representation.

Given a (multi)set variable V drawn from some multiset S, it is possible to construct a CSP where V is replaced by a new variable V' where $S - V = V'$, and all the constraints are altered accordingly. This has the property that any constraints which were *monotonically set increasing* on V will be *monotonically* *set decreasing* on V', and vice versa. This does not unfortunately solve all the difficulties related to the *explicit* representation. Firstly given a (multi)set of small maximum size drawn from a large set then this inversion process may generate a variable which has too large a domain to be usable in practice. Also Theorem 5.14 does not help in the case where a problem has both *monotonically set increasing* and *monotonically set decreasing* constraints, or constraints which satisfy neither condition.

5.5 Conclusion

This chapter has shown a number of results involving dominating representations. For variable representations a number of methods of identifying or generating dominating representations have been presented, and a number of non-trivial dominance relations between the *occurrence*, *explicit* and *Gent* representations have been presented. It has been shown that literal equivalent representations have the same representative power. The results of previous papers which showed a difference between literal equivalent representations were shown to involve either limitations of the solver or bad modelling, both of which could be overcome by solver improvements without having to introduce multiple literal equivalent representations.

Two special cases were considered on the standard *explicit* representation, which being non-simple is difficult to study. Firstly the n-way limited *explicit* representation was presented. While still not simple, this limited version of the *explicit* representation is much easier to study and occurs frequently in practice. Secondly, monotonic set constraints were considered. These constraints help to demonstrate in which cases the *explicit* representation performs well and also occur in practice.

Chapter 6

Perfect Representations

Dominance provides a powerful method of comparing representations, but in many cases it is too coarse, as it compares all possible CSPs. Two representations may be incomparable when considered over all CSPs, but one may be better than the other with respect to a single CSP. Comparing two representations on an arbitrary single CSP without solving it is unfortunately not possible at present.

This difficulty has arisen before in the study of CSPs, in particular when looking at CSPs which can be solved in polynomial time. One way this problem is solved is by splitting CSPs into classes. This is done by considering all CSPs whose constraints come from a certain language [10]. This categorisation of CSPs will be considered in this chapter.

6.1 Perfect Constraint Families

The aim of this chapter is to answer the question whether a particular representation is as good as the complete representation on a particular language of constraints, which will be used to express both the basic problem and the branching. This is formally given in Definition 6.1. While this does not allow two representations to be directly compared, in the case where a representation is equivalent to the complete representation, it must dominate all others.

Definition 6.1. Given:

- A set of constraints S over an array of variables $\overline{V}$
- An array $\overline{R}$ where $R[i]$ is a representation of $V[i]$

- A propagator $rep(s)$ for each $s \in S$ on $\overline{R}$

Then $\overline{R}$ and the $rep(s)$ are a **perfect GAC** implementation of S if the fixed point of GAC propagators on the members of S allow the same set of assignments as the fixed point of the representations of the constraints.

Definition 6.1 only considers the case where propagation begins from all sub-domains containing all values. Assuming the set S has the branching constraints added to it during search, then this captures all possible states that can occur during search. However, it makes it hard to combine sets of constraints, as it assumes no other propagation occurs. Example 6.1 provides an example of how a list of representations can be a perfect GAC implementation for each of two sets of constraints, but not of their union.

Example 6.1. *Consider a CSP P with two multiset variables X and Y, both with the domain $\mathbb{P}(\{1,2,3,4,5\})$*[1] *and the two constraints $\mathbf{2} \notin \mathbf{Y}$ and $|\mathbf{X}| \leq |\mathbf{Y}|$.*

Generate a new CSP P' where X and Y are represented with the occurrence representation by arrays of variables X' and Y', with the two constraints mapped to $\mathbf{Y}'[\mathbf{2}] \neq \mathbf{1}$ and $\sum_\mathbf{i} \mathbf{X}'[\mathbf{i}] \leq \sum_\mathbf{i} \mathbf{Y}'[\mathbf{i}]$.

Beginning from all sub-domains containing every possible value, after propagating $\mathbf{Y}'[\mathbf{2}] \neq \mathbf{1}$ all sets which are still represented by $\overline{Y'}$ are allowed by the constraint, and therefore this constraint is perfect. Similarly propagating the constraint $\sum_\mathbf{i} \mathbf{X}'[\mathbf{i}] \leq \sum_\mathbf{i} \mathbf{Y}'[\mathbf{i}]$ from initial domains is also perfect.

However if $\mathbf{Y}'[\mathbf{2}] \neq \mathbf{1}$ is propagated, followed by $\sum_\mathbf{i} \mathbf{X}'[\mathbf{i}] \leq \sum_\mathbf{i} \mathbf{Y}'[\mathbf{i}]$, then no values are removed from $\overline{X'}$. This means the set $X = \{1,2,3,4,5\}$ is still represented despite the fact it does not satisfy the constraint $|\mathbf{X}| \leq |\mathbf{Y}|$, as $Y'[2] = 0$ and so X cannot be a set with all 5 elements.

Example 6.1 shows that having two sets of perfect constraints does not mean their union will be perfect. Therefore while it is a much stronger condition, it is useful to consider the case where propagation is "perfect" when begun from any set of sub-domains, as given in Definition 6.2. While this property is much stronger, it will be shown that it is much better behaved in practice, and satisfied by many constraints.

Definition 6.2. Consider a constraint C over an array of variables $\overline{V}$, an array $\overline{R}$ where $R[i]$ is a representation of $V[i]$, and a propagator $rep(P)$ for C on $\overline{R}$,

[1]This can of course also be considered as a set variable

then $\overline{R}$ and $rep(P)$ are a **completely perfect GAC** implementation of C if the sub-domains of any representational state of a fixed point of P are *GAC* with respect to C.

Note that both Definition 6.1 and Definition 6.2 consider applying a list of representations to a list of variables, rather than applying a single representation to a single variable. This is because in many cases replacing *all* variables of the same type is perfect, whereas replacing *a single* variable with a representation is not.

Note that a completely perfect GAC representation does **not** require that every sub-domain of the original variables which are GAC are reachable. It only requires that given a representational state, after propagation the representation state reached is GAC.

Example 6.2. *Consider a variable A with domain $\mathbb{P}(\{1,\ldots,n\})$ represented with the occurrence representation with occurrence array $\overline{V}$. The constraint $\mathbf{a} \in \mathbf{A}$ for a constant value a is represented by the constraint $\mathbf{V[a] = 1}$. Any sub-domain of $\overline{V}$ which is GAC with respect to this constraint will clearly have $V[a]$ with only* 1 *in its domain and therefore any assignment to $\overline{V}$ which represents an element of A will satisfy the original constraint. The occurrence representation is therefore completely perfect GAC with respect to the constraint $\mathbf{a} \in \mathbf{A}$.*

Example 6.3. *Given a natural number n, consider the CSP containing a single set variable A drawn from $\{1,2,\ldots,2n\}$ and the two constraints $|\mathbf{A}| = \mathbf{n}$ and $|\mathbf{A}| = \mathbf{n+1}$. GAC propagation on these two constraints would empty the domain of A. Now consider A represented by the occurrence representation with occurrence array $\overline{V}$. The two original constraint will map to $\mathbf{sum}(\overline{\mathbf{V}}) = \mathbf{n}$ and $\mathbf{sum}(\overline{\mathbf{V}}) = \mathbf{n+1}$. On any sub-domain of V where less than $n-1$ variables are instantiated, neither of the constraints would remove any values from the domains of any variables if propagated, so the size of the search tree using any binary branching method with constraints of the type $X = a, X \neq a$ will be at least 2^{n-2} nodes.*

Example 6.2 gives an example of a completely perfect GAC constraint, while Example 6.3 gives an example of a non-perfect one, which leads to an exponential increase in search size compared to using the complete representation. Lemma 6.1 shows that if the initial set of constraints along with any set of branching constraint are completely perfect GAC, then the search which results from using the representations is identical to using the complete representation.

Lemma 6.1. *Given a CSP implementation, if both the original set of propagators, and the sets formed by unioning these propagators with the set of propagators used for branching at any node of search, are completely perfect GAC with respect to a set of representations, then applying those representations will lead to an identical search to the case of not using these representations at all.*

Proof. This result is true by definition. In any node of search, the result of propagating both the branching constraints and the original constraints must result in the same allowed set of assignments whether or not the representations are applied. In particular if no assignments are allowed in the original CSP, no assignments are allowed once the CSP is represented and therefore search will backtrack. □

Given that a set of representations being perfect, or completely perfect, with respect to a set of constraints is useful, it would be useful to construct methods of easily finding such sets. In theory it is sufficient to check the constraints against the definition, but this can be difficult, particularly in the case of completely perfect constraints. In the special case of variable representations, there are a number of interesting special families of constraints which are easy to identify and commonly occurring, which can be identified as completely perfect GAC.

The rest of this section will give the definitions and lemmas required to prove Theorem 6.4, which shows an important sufficient condition for a complete simple variable representation to be completely perfect GAC with respect to a given constraint.

One commonly occurring property, given in Definition 6.3, is that a single constraint can be flattened into a set of constraints with no overlap in their scopes, called a split. Recognising such splits is useful, as the smaller constraints can often be easier to work with. Example 6.4 gives an example of such a split. The split of this constraint is clear from the way in which it is written and in practice often the most compact way of expressing a constraint makes the split clear.

Definition 6.3. Given a constraint c, a set of constraints S is defined to be a **split of** c if the scopes of any pair of elements of S are disjoint, the scope of c is equal to the union of the scopes of the elements of S and given an assignment to the variables in $scope(c)$, c is true if and only if all of the elements of S are.

Example 6.4. *Consider two set variables X and Y, which are represented by the occurrence representation with arrays $\overline{X'}$ and $\overline{Y'}$. Then the constraint $X \subseteq Y$ is* *mapped to the single constraint* $\forall i.\ X'[i] \implies Y'[i]$*, which can clearly be split into a set of constraints of the type* $X'[i] \implies Y'[i]$*, one for each value which can be assigned to* i*.*

While there may be a number of different splits of a particular constraint, on any particular set of scopes, Lemma 6.2 shows that there is a well-defined minimal set of constraints. By minimal we mean that each constraint in the split contains only allowed assignments which must be allowed by that constraint in any set of constraints in the same split. Having this well-defined minimal set of constraints will make later proofs simpler.

Lemma 6.2. *Consider a split* S *of constraint* C*. Then the set of constraints* S' *generated by creating a new constraint* s' *for each* $s \in S$*, with* $scope(s') = scope(s)$ *and an assignment allowed by* s' *if it can be extended to a valid assignment of* C*, is also a split of* C*. Further, all of the tuples in these constraints must have been in* S*.*

Proof. There is no assignment to $scope(C)$ which is allowed by C and not by all the constraints in S' as each $s' \in S'$ accepts any assignment which can be extended to a complete assignment of $scope(C)$ which satisfies C.

Consider a constraint $s \in S$, and its equivalent $s' \in S'$. The constraint s cannot allow less assignments than s', as doing so would forbid some solution to C. Therefore the set of solutions to the intersection of all constraints in S' must be a subset of the solutions to the intersection of all constraints in S. Therefore the solutions to S' are a subset of the solutions to S, which are equal to the solutions to C.

If any tuples were deleted from any of the constraints in S', then clearly there would be a solution of C which was not a solution of S', as S' contains exactly those tuples which extend to a solution. Therefore S' and C have the same set of solutions. □

The following technical proposition gives an alternative definition of a completely perfect GAC implementation just on variable representations.

Proposition 6.3. *Consider an array of CSP variables* $\overline{X}$ *and a constraint* S*, and an array of variable representations* $\langle\overline{V_i}, f_j\rangle$ *where* $\langle V_i, f_i\rangle$ *represents* $X[i]$ *and a propagator* P *on the* $\overline{V_i}$ *which represents* S*. Then* P *is a completely perfect GAC implementation of* S *if and only if given any sub-domains of the variables of each*

$\overline{V_i}$ which are fixed under P, any assignment $\overline{v_j}$ to the variables to any specific V_j such that $f_j(v_j)$ can be extended to an assignment which satisfies S.

Proof. The definition of completely perfect GAC implementation considers only the representation states achieved after propagation has finished, and therefore only the fixed points of P must be considered.

Consider an array $\overline{r}$ where $r[i]$ is a representational state of $\overline{V_i}$, and $P(\overline{r}) = \overline{r}$. Construct the array $\overline{A}$ where $A[i]$ is the set of assignments allowed by $r[i]$. Now $\overline{A}$ is GAC with respect to S exactly if any assignment to any sub-domain in $\overline{A}$ is extendable to a solution. This is exactly equivalent to any assignment to any of the $r[i]$ where $f_i(r[i])$ is defined is extendable to a solution. □

The results from this section so far are used to prove Theorem 6.4. Using the existence of splits, this Theorem provides a powerful sufficient, but not necessary, condition which can be used to demonstrate that constraints on variable representations have completely perfect GAC implementations.

Theorem 6.4. *Given an array of simple variable representations $\overline{R}$ of a list of variables $\overline{V}$, a constraint c on $\overline{V}$ and the image c' of c under $\overline{R}$, there exists a completely perfect GAC implementation of c if there exists a split S of c' where each $s \in S$ contains at most one variable from any element of $\overline{R}$.*

Proof. If there exists a valid split of c' into a set of constraints S each of whose elements contains at most one variable from any element of $\overline{V}$, then there is a split of the form given in Lemma 6.2. Without loss of generality assume any split takes this form.

Given sub-domains $\overline{r}$ of each element of $\overline{R}$, by Proposition 6.3 it is sufficient to show any assignment to any $r[i]$ which represents an assignment to $V[i]$ can be extended to a satisfying assignment of c'.

Assume without loss of generality some assignment a to r_1 fails this requirement. This assignment must represent an assignment to the represented variable, and therefore as S is logically equivalent to c' in the context of the representational constraints, it must fail to be extendable to a solution to S. As the elements of S are each on a disjoint set of variables, it must fail to satisfy some specific $s \in S$ and therefore fail to be extendable to a solution to a specific element of $s \in S$. However, as this is only a single variable in s, and the current sub-domains are GAC with respect to s so this is a contradiction. □

Corollary 6.5. *Any constraint on an array of unsized multiset variables $\overline{V}$ of length n, each with domain drawn from a set S which can be specified in the form* $\forall \mathbf{s} \in \mathbf{S}.\ \mathbf{p}(\mathbf{occ}(\mathbf{s}, \mathbf{V}[\mathbf{1}]), \ldots, \mathbf{occ}(\mathbf{s}, \mathbf{V}[\mathbf{n}]))$ *any predicate p is completely GAC perfect where all elements of $\overline{V}$ are represented with the occurrence representation.*

Proof. The term $occ(s, V[i])$ in the *occurrence* representation is represented by $V'_i[s]$, where $\overline{V'_i}$ is the *occurrence* vector representing $V[i]$. There is therefore a natural split of the constraint, where in each element of the split the s^{th} element of each of the *occurrence* representations is used. This is demonstrates the split required by Theorem 6.4. □

Corollary 6.5 demonstrates a large family of constraints with completely perfect GAC implementations on the *occurrence* representations. It demonstrates any constraint built from the operators $\cup, \cap, -, +, =, \subseteq, \triangle$[2] on unsized sets and multisets has a completely perfect GAC implementation for the *occurrence* representation. Further, the constraints $x \in S, x \notin S$ and $occ(x, S) = i$ for fixed i can also be seen to have a completely perfect GAC implementation by Booleanizing x (Definition 5.8).

The *explicit* representation does not have a completely perfect GAC implementation for any of these constraints except $\notin$. This can be seen as this is the only one of these constraints which will split on the *explicit* representation. This is also the only one of these constraints on which the *Gent* representation has a completely perfect GAC implementation.

One notable missing constraint from the list of constraints on which the *occurrence* representation is completely perfect GAC is $|\mathbf{S}| = \mathbf{c}$, for either fixed or variable c. The lack of a split for this constraint is clear, and Example 6.1 showed explicitly that this constraint is not completely perfect.

One possible way around this would be to consider the *occurrence + size* representation (given at the end of Section 4.5), on which $|\mathbf{S}| = \mathbf{c}$ clearly has a split which would satisfy Theorem 6.4, as the constraint would only refer to the size variable. However, on the *occurrence + size* representation, Corollary 6.5 no longer holds. While $\in$ and $\notin$ are still completely perfect GAC, most of the other constraints, including in particular $\mathbf{A} \cup \mathbf{B} = \mathbf{C}$ and $\mathbf{A} \cap \mathbf{B} = \mathbf{C}$, do not have a completely perfect GAC implementation on the *occurrence + size* representation.

From the definition of perfect GAC implementation, it is clear that if one list of representations has a perfect GAC implementation (**not** completely perfect

[2]symmetric difference

GAC) on a particular set of constraints, then replacing one or many of these representations with one which dominates them will produce a new list of representations on which there will exist a perfect GAC implementation. Therefore as expected on CSP where the *occurrence* representation is completely perfect on both the CSP and the branching constraints, the *occurrence + size* representation will be a perfect GAC implementation and therefore still achieve the minimal (and identically sized) search tree. However, in general they will not be completely perfect on the *occurrence + size* representation.

This still does not help with solving CSPs that contain set variables which have constraints on them that both restrict their size and also impose other constraints such as subset or intersection. On such CSPs neither the *occurrence* representation or the *occurrence + size* representation have a completely perfect GAC implementation. This problem will be discussed in Section 6.3.

6.2 Decomposing and Rewriting Constraints

Many constraint problems are most naturally specified with constraints built recursively from a list of operators, for example $(\mathbf{A} \cup \mathbf{B}) \cup (\mathbf{C} \cap \mathbf{D}) \subseteq \mathbf{E}$. Rather than attempt to implement propagators for arbitrarily complex constraints, CSP solvers define a set of primitive constraints and then break all more complex constraints down into these primitive constraints using extra variables. Example 6.5 gives a simple example of flattening. Flattening a constraint is very different to splitting, as flattening introduces extra variables.

Example 6.5. *One flattening of the constraint $(A \cup B) \cup (C \cap D) \subseteq E$ over variables A, B, C, D, E into smaller constraints is given below, using three new variables W, X and Y.*

- $A \cup B = W$
- $C \cap D = X$
- $W \cup X = Y$
- $Y \subseteq E$

In many constraint solvers, flattening is done internally and is not visible to the user. In the case given in Example 6.5, the newly introduced set variables may be implemented using the *occurrence* representation, meaning a large number of CSP variables are introduced. This section looks at how representations and flattening interact. In particular, it will be shown that if the constraints both before and after flattening have a completely perfect GAC implementation with respect to some set of representations, then flattening does not affect the size of search. While flattening is used in many CP solvers, the author of this thesis has been unable to find any proofs of its effect on search. Therefore in the process of proving results about flattening in the presence of representations, results about flattening in a traditional CP context will also be presented.

In general there are some complications with attempting to automate the process of flattening a constraint. The type of each new variable and a domain large enough to hold any value it could take must be calculated. Allowing too large a domain for any introduced variable would be correct but would reduce efficiency. Also, the set of constraints that a CSP solver implements may be large and provide more than one way of splitting a particular constraint, with their own trade-offs and inefficiencies. These issues will not be dealt with directly in this thesis. A further discussion of such issues was involved in the design of the ESSENCE language, and CONJURE refinement system, and the recent further mapping of ESSENCE' to the Minion constraint solver [68], which handled mapping ESSENCE' into Minion's limited input language. The arithmetic part of Minion's language is provided in Example 6.6.

Example 6.6. *The Minion CSP solver, as of version 0.4, requires all arithmetic constraints are flattened into constraints from the following list. In these constraints the x_i represent variables, c and the d_i represent constants and S represents any finite set of integers.*

1. $\sum_{i \in S} x_i \leq c$	*2.* $\sum_{i \in S} x_i \geq c$
3. $x_1 \leq x_2 + c$	*4.* $\sum_{i \in S} d_i x_i \leq c$
5. $\sum_{i \in S} d_i x_i \geq c$	*6.* $x_1 \neq x_2$
7. $x_1 \times x_2 = x_3$	*8.* $x_1^{x_2} = x_3$
9. $x_1 / x_2 = x_3$	

Constraints 1 to 4 in this list are redundant as they could all be implemented using the 5th one, but are provided to give better efficiency. The obviously missing constraint $\sum x_i = c$ must be implemented using $\sum x_i \leq c$ and $\sum x_i \geq c$.

Studying how flattening affects search is useful for two reasons. First of all it provides a method of analysing large families of constraints. Second it is the way in which these constraints are commonly implemented and therefore provides results which can be usefully applied directly to existing solvers. Flattening has previously been studied for the special case of constraints made from the operators $+, \times, \leq, =$ and $\neq$ on integer variables by Harvey and Stuckey [40]. In this chapter this study will be extended to consider both arbitrary constraints and more importantly taking into account any representational choices. Examples 6.7 and 6.8 provide two concrete examples of flattening.

Example 6.7. *Given the constraint* $\mathbf{A+B+C=0}$ *where* A, B *and* C *have domain* $\{-1, 0, 1\}$*, one flattening is generated by the new variable constrained to be* $\mathbf{X=A+B}$ *and the constraint* $\mathbf{X+C=0}$*, where* X *is a new variable with domain* $\{-2, 1, 0, 1, -2\}$*.*

Example 6.8. *Given the constraint* $\mathbf{A+B*A=0}$ *where the domain of* A *is* $\{-n, \ldots, n\}$ *and* B *is* $\{-1, 1\}$*, one valid flattening is generated by introducing a new variable* X *with domain* $\{-n, \ldots, n\}$ *and using the constraints* $\mathbf{X=B*A}$ *and* $\mathbf{A+X=0}$*.*

This flattening can cause the size of search to increase as A *occurs in both constraints. In the original constraint, if* B *is assigned* 1 *then GAC propagation causes* A *to be assigned* 0*. In the split version, if* B *is assigned* 1 *then the domain of neither* X *or* A *is propagated by GAC propagation on either constraint in isolation.*

Example 6.7 provides an example of a flattening which will not increase the size of search. This fact will be proved later in this chapter. Example 6.8 shows a case where a split causes a loss of propagation compared to the constraint before decomposition. In this case it is because one of the original variables ends up in both new constraints. Definition 6.4 formalises flattening a constraint.

Definition 6.4. A **flattening** of a constraint C is defined by:

- An array of variables $\overline{V}$, each element of which is contained in $\mathit{scope}(C)$
- An array of new variables $\overline{X}$
- A function $f : \overline{V} \to \overline{X}$
- A constraint C' with $\mathit{scope}(C') \subseteq \mathit{scope}(C) + \overline{X}$

Where each assignment to $\mathit{scope}(C)$ satisfies C if and only if it can be extended to an assignment to $\mathit{scope}(C) \cup \overline{X}$ which satisfies $C' \wedge \big(f(\overline{V}) = \overline{X}\big)$. If such an assignment to $\overline{X}$ exists, it must be unique. A flattening is **pure** if no element of $\overline{V}$ is in $\mathit{scope}(C')$.

Replacing a constraint with a flattened version leads to a new CSP with the same number of solutions as the original, with each solution to the original CSP extended by a unique assignment to the newly added variables. However, the search spaces can be very different. One obvious cause for this is that a number of extra variables have been introduced. Consider the case where these new variables are ignored by any variable heuristic and left to the end of the search. As the introduced variables are always contained on the right hand side of a constraint of the type $f(\overline{V}) = \overline{X}$, if they are left until the end of search there will be at most a single value each of them can take. Performing GAC propagation on the constraint $f(\overline{V}) = \overline{X}$ will either assign them a single value or empty their domains.

Apart from this, the other way in which flattened constraints can effect search is by causing a reduction (or increase) in the propagation of the original variables. Example 6.8 provides an example of where this can occur. In general this problem is related to the decomposition being pure. In the case where the decomposition is not pure, as seem in Example 6.8 propagation can be lost, caused by multiple variables occurring in both constraints.

In the case where a split is pure and only a single variable is introduced, then the CSP after the split is applied will have an identical search tree, if the new variables are put at the end of the search tree and the newly introduced constraints and original constraint are propagated to GAC. This result is not directly related to representations but is a generally useful result for implementing constraint solvers and will be proven as a special case of the more general result about flattening constraints in the presence of representations, given in Theorem 6.6

In the case where more than one variable is introduced, in general even a pure split will cause a loss of propagation. One important case for this thesis, as given in Example 6.5, is where the introduced variables are a representation.

The particular case which will be closely studied here is when the introduced variables are a variable representation of a single variable. In this case, if the representations used have completely perfect GAC implementation on the newly introduced variables on both constraints after the split, no propagation will be lost and the search will be the same. To prove this result, the technical Theorem 6.6 must first be proved. While this does not directly refer to representations, the conditions it places on the variables in the split are closely related to the requirements of a completely perfect GAC implementation of a constraint on a representation.

Theorem 6.6. *Consider a constraint C where scope(C) contains the array of variables $\overline{V}$, which can be pure flattened as:*

- *A function $f : \overline{V} \rightarrow \overline{X}$*
  The constraint $f(\overline{V}) = \overline{X}$ will be refered to as C_f
- *A constraint C' where scope$(C') =$ scope$(C) - \overline{V} + \overline{X}$.*

Assume that:

1. *Any assignment to $\overline{X}$ which is not in the range of f appears in no valid assignment of C'.*
2. *Given a list of sub-domains of the variables in either scope(C_f) or scope(C') which are GAC, then any assignment to $\overline{X}$ contained both in that sub-domain and in the domain of f can always be extended to a valid assignment.*

In this case, given a sub-domain of each of the variables in scope(C), they satisfy $GAC(C)$ if and only if there exists a sub-domain of the variables in $\overline{X}$ such that the sub-domains of scope(C) and $\overline{X}$ together satisfy $GAC(C_f) \wedge GAC(C')$.

Proof. Given an array of sub-domains D_C for the variables in *scope*(C), an array of sub-domains $D_{\overline{X}}$ of the variables in $\overline{X}$ can be constructed by taking the union of all assignments to $\overline{X}$ generated by taking $f(\overline{v})$ for all assignments to $\overline{v}$ in D_C which can be extended to a valid assignment of *scope*(C).

If D_C satisfies $GAC(C)$, then D_C and $D_{\overline{X}}$ together will satisfy $GAC(C_f) \wedge GAC(C')$ because any assignment to D_C which satisfied C by definition of $D_{\overline{X}}$ can be extend in the sub-domain of $D_{\overline{X}}$ to an assignment which satisfies $C_f \wedge C'$. Further, each value in the sub-domain of D_X was added only because it was part of an assignment which satisfied the constraint.

If D_C does not satisfy $GAC(C)$, consider if there did exist a sub-domain of the elements of D_X such that D_C and D_X together satisfy $GAC(C_f) \wedge GAC(C')$. There must exist a variable $z \in$ *scope*(C) and assignment a to z which cannot be extended to a valid assignment of C in these sub-domains. There are two cases to consider, only one of which can be true as the only overlap of these scopes is on $\overline{X}$, which is disjoint from *scope*(C) as the decomposition is pure

$\mathbf{z \in scope(C_f)}$**:** As $GAC(C_f)$ is true, there must be an assignment to $\overline{V}$ and $\overline{X}$ with $z = a$ which satisfies C_f. The assignment this gives to $\overline{X}$ must be in the domain of f, as it satisfies C_f. This assignment to $\overline{X}$ must be extendable to an assignment to C', as this was a requirement on C'. This gives assignments to C_f and C' which satisfy both constraints and agree on their overlap, which is only $\overline{X}$ as the split was pure. Therefore this assignment satisfies $\mathbf{C'} \wedge \mathbf{C_f}$, and therefore C.

$\mathbf{z} \in$ scope($\mathbf{C'}$): As $GAC(C')$ is true, then there must be an assignment to the variables of C' which includes $z = a$ which satisfies C'. This assignment includes an assignment to $\overline{X}$ which must be in the domain of f by requirement, and so there must be an assignment to $\overline{V}$ such that C_f is satisfied. This gives assignments to the scopes of both C_f and C' which satisfy both constraints and agree on their overlap, so this assignment satisfies $\mathbf{C'} \wedge \mathbf{C_f}$, and therefore C.

□

Theorem 6.6 does not directly apply to representations. However, the condition it places on the array of overlapping variables, denoted $\overline{X}$, is exactly the condition that the overlapping variables are the representation of a single variable that has a completely perfect GAC implementation. This leads to Corollary 6.7.

Corollary 6.7. *Consider a pure flattening of a constraint C by introducing a single variable x and the constraints C' and the function $f(\overline{V}) \rightarrow x$ for some $\overline{V}$ in scope(C). Given representations for each variable in scope(C) and for x with a completely perfect GAC implementation in both the constraints C' and $f(\overline{V}) \rightarrow x$, then replacing the variables with the representations will not lead to an increase in search.*

Proof. Follows from Theorem 6.6, as after the representation is applied, the representation of x must satisfy the conditions required in that theorem. □

Using the results from Section 6.1 and Corollary 6.7, the major result this gives is that any constraint built from the operators on which the *occurrence* representation is perfect, which include $\cup, \cap, =, \subseteq$ and $\in$, on unsized sets and multisets have a completely perfect GAC implementation on the *occurrence* representation, which still holds after the constraints are flattened.

The major limitation of the occurrence representation this does not address it that it does not allow a completely perfect GAC implementation of cardinality constraint. The next section investigates this problem and will place a number of strong limitations on any representation which are completely perfect for the cardinality constraint.

6.3 NP-Hard Perfect Representations

Section 6.1 showed that the *occurrence* representation has a completely perfect GAC implementation of a large range of common set and multiset constraints. One notable exception is constraints which restrict the size of the (multi)set. An obvious aim would be to search for an efficient representation which has a completely perfect GAC implementation of constraints which restrict the size of a (multi)set and also of other commonly occurring (multi)set constraints. This section will show that such representations do not exist.

To show this, it will be shown that any implementation which did have a completely perfect GAC implementation on both the cardinality constraint and a small number of other common (multi)set constraints would provide a method of solving an NP-complete problem in a polynomial sized search tree. This means at least one constraint must require exponential time to propagate, assuming $P \neq NP$.

One famous class of NP-complete problems is SAT. This was given previously in Definition 2.14 on page 21, along with 1-in-k SAT, which will also be used here. As discussed in Section 2.2.1, SAT was one of the first problems to be shown NP-complete. Here, SAT and some specialisations of it will be used to prove that any completely perfect GAC implementations of some constraints must be NP-hard on any representation of sets or multisets.

One obvious question would be are there representations which are completely perfect GAC on both size constraints and also some or all of the other common set constraints and which can be implemented in polynomial space and time. Clearly the complete representation allows these constraints to be implemented as completely perfect GAC, but requires worst case exponential space and time to implement propagators. Such a representation would be very useful. Unfortunately how to build one is not obvious, it cannot for example be achieved simply by channelling together multiple representations which individually have completely perfect GAC of some of the constraints in question.

Many small sets of constraints including the cardinality constraint have the property that any representation which has a completely perfect GAC implementation of all the constraints must have one propagator which is NP-hard. These families include just the constraint $|S| = c$ for a constant c and either of the constraints $A \cup B = C$ or $A \cap B = C$. This will be proven by showing that using such constraints it is possible to build a family of CSPs which are themselves NP-complete, but which can be solved in a search tree of polynomial size.

A number of classes of CSPs which require only polynomial sized search tree to solve if constraints are propagated to GAC have been identified. Identifying the class which will be used here requires first defining the constraint graph of a CSP, given in Definition 6.5. Figure 6.1 provides an example of a constraint tree. In this diagram the small black circles are nodes, one for each variable of the CSP, and the large ellipses represent edges, each containing all the variables contained in the scope of one constraint.

Definition 6.5. A **hyper-graph** is a pair $\langle V, E \rangle$ where V is an arbitrary set, called the nodes of the hyper-graph, and E is a set of subsets of V, called the edges. Hyper-graphs differ from traditional graphs in that each edge can contain an arbitrary number of vertices.

The **constraint hyper-graph** of the CSP is a hyper-graph which contains one node for each variable and one edge for each constraint, where the nodes in the edge are the variables in its scope.

Freuder proved [24] that where the constraint graph of a CSP has certain properties, then the CSP can be solved without backtrack if the variables are assigned in a certain order and GAC propagation is achieved. An exact copy of this result is given in Lemma 6.8. Notice that the example in Figure 6.1 satisfies this property.

Lemma 6.8. *Consider a CSP $P = \langle V, D, C \rangle$ where the intersection of the scopes of any pair of constraints contains at most one variable and the constraint graph of the CSP is a tree. Then there exists a variable ordering where a backtracking search which GAC propagates all constraints will either find a solution or prove no solution exists in $|V|$ search nodes.*

Proof. Proved in [24]. □

Given Lemma 6.8, the aim is now to attempt to construct problems which satisfy this tree condition and whose solution would solve an NP-complete problem. As such problems can be solved in a linear number of search nodes, this would prove that at least one of the propagation algorithms used in the problem must itself be NP-hard.

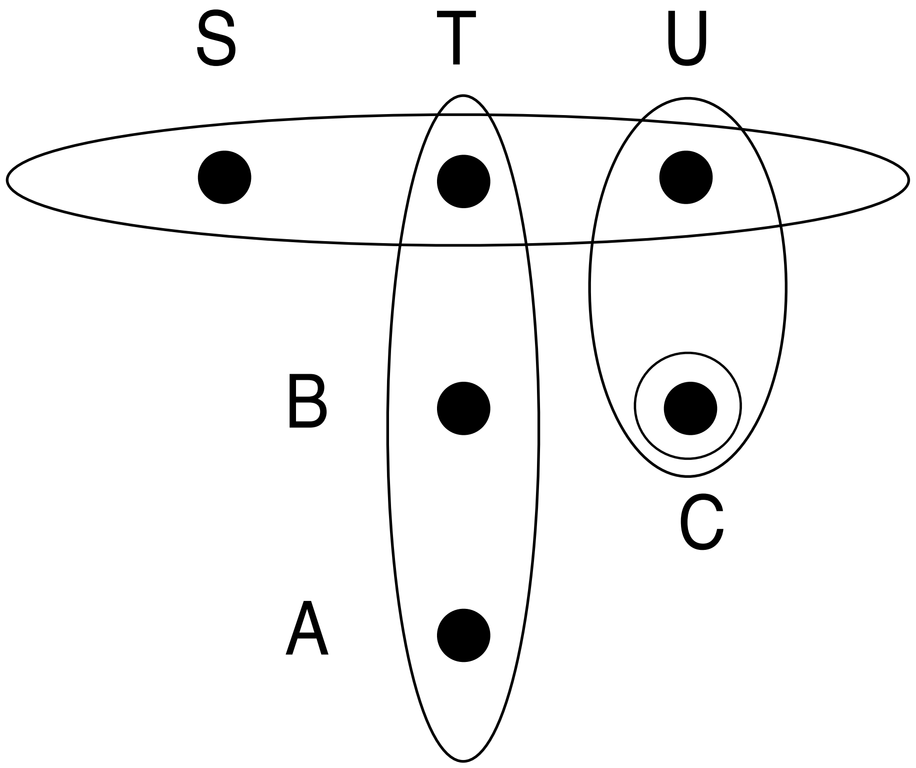

Figure 6.1: The constraint hyper-graph of: $S \cap T \subseteq U, A \cup B \subseteq T, C \subseteq U, |C| = 2$

The basic aim of this section is to map 1-in-k SAT to a problem which satisfies the requirements of Lemma 6.8. These constructions will be similar to those presented in [6]. However, there are further requirements here than it that paper, as not only must the mapping solve 1-in-k SAT, it must also satisfy the requirements of Lemma 6.8.

Clearly if such a mapping can be built, and then shown to satisfy the conditions of Lemma 6.8, then it will have been proven that no representation can have a completely perfect GAC implementation of all the constraints used and also still be solved in time polynomial in the size of S. Implementing the framework above requires finding a method of specifying that exactly one value from some subset of V occurs in S. If this can be implemented in such a way that it satisfies Lemma 6.8, then this will prove the set of constraints used can not have a polynomial time completely perfect GAC implementation for any representation, if $P \neq NP$.

Lemma 6.9. *Consider a set variable X whose domain is all subsets of some set s and any fixed subset t of s. Then using two set variables with the same domain as X and only constraints of the form:*

1. $|A| = c$ for fixed c and a set variable A

2. $A \cap B = C$ for set variables A, B and C

3. $A \subseteq B$ for set variable A and fixed set B

it is possible to implement the constraint "S contains exactly one element of t" in a way which satisfies the conditions of Lemma 6.8.

Proof. Consider the construction:

1. Two set variables, $Check_t$ and Val_t whose domain is all subsets of s.

2. The constraints $|Check_t| = 1$, $|Val_t| = |t|$, $Val_t \subseteq t$ and $S \cap Val_t = Check_t$.

In all solutions Val_t must be assigned t. Any assignment to S which allows exactly one element of s can be extended to a valid assignment to both Val_t and $Check_t$ by assignment $Check_t$ the set containing only this value. In any other case, there is no assignment to $Check_t$ which will will satisfy all the constraints. □

Corollary 6.10. *Given a representation which is completely perfect on the constraints $A \cap B = C$, $|A| = c$ and $A \subseteq B$ for set variables A, B and C and constant c, the propagators of at least one of these constraints must be NP-hard.*

Proof. Lemma 6.9 shows how these constraints can be used to form the constraint "S contains exactly one element of t" for set variable S and constant set t in a way which satisfies Lemma 6.8. The following gives a description of how to implement 1-in-k SAT for a set of variables V and a set of clauses C.

1. A single set variable S which contains all the literals of V
   The assignment to S will express the solution to the SAT instance.

2. One constraint for each variable v in V which imposes that exactly one of the literals of v is contained in S.

3. One constraint for every clause in C which imposes that at exactly one literal in the clause is in S.

Each of these constraints can be expressed as "S contains exactly one element of t" and are independent, so using Lemma 6.9 to impose each one will satisfy Lemma 6.8. This will give a construction which solves 1-in-k SAT in a backtrack free search, and therefore one of the propagators of these constraints must be NP-hard. □

Corollary 6.10 shows that a representation having a completely perfect GAC implementation of just the constraints $A \cap B = C$, $|A| = c$ and $A \subseteq S$ for set variables A, B and C, and integer constant c and set constant S, is enough to construct 1-in-k SAT. This shows that only a very small number of common set constraints added to $|A| = c$ create a set where it is NP-hard to have representation with complete perfect GAC implementations of all the constraints. This set of constraints is not as simple as can be achieved, for example Lemma 6.11 shows how the need for intersection can be eliminated.

Lemma 6.11. *Given a set variable S whose domain is all subsets of a set s and a fixed subset t of s. Then using two set variables whose domain are also all subsets of s and the constraints:*

1. *$|A| = c$ for set variable A and constant c.*
2. *$A \subseteq B$ for set variables A and B.*
3. *$A \subseteq B$ for set variable A and constant set B.*

it is possible to implement the constraint "S contains exactly one element of s" in a way which satisfies the conditions of Lemma 6.8, and can therefore be used in Corollary 6.10.

Proof. Consider the construction:

1. Two set variables T_1 and T_2 whose domain is all subsets of s.
2. The constraint $|T_1| = |s - t| + 1$.
3. The constraint $(s - t) \subseteq T_1$.
4. The constraint $S \subseteq T_1$.
5. The constraint $|T_2| = 1$.
6. The constraint $T_2 \subseteq s$.
7. The constraint $T_2 \subseteq S$.

T_1 must contain all elements of $s - t$ and is of size $|s - t| + 1$, so contains exactly one member of t. As $S \subseteq T_1$, S can contain at most one element from t. T_2 is of size exactly 1 and as $T_2 \subseteq t$, that single value must lie in t. As $T_2 \subseteq S$, that value must also lie in S.

These two results together show S must contain exactly one element of t. As there are no constraints between T_1 and T_2, the constraints form a tree. □

6.4 Conclusion

This chapter has shown that for a large number of set constraints, the occurrence representation performs as well as the complete representation and continues to do so when flattened. This provides a powerful and useful result to modellers. Furthermore, there is no representation which has a completely perfect GAC implementation on both the cardinality of a set and on most other common set constraints where these propagators run in polynomial time, unless $P = NP$. This does not mean that such representations are not useful, for example ROBDDs [50] have an NP-complete propagation algorithm yet are still competitive on many problems. It does however give a strong theoretical limit on the complexity of any implementation which has a completely perfect GAC implementation of these constraints.

Chapter 7

Random Representations

So far in this thesis the representations considered have had a natural structure which allows them to be expressed in a compact intensional form. There is however, a very large number of representations which do not have a natural compact format and lack any obvious structure. In this chapter a number of experiments are performed to compare randomly generated representations with the *occurrence* representation.

These experiments are **not** intended to provide comprehensive testing of all representations in this thesis, rather to illustrate a few important concepts. A careful practical comparison of many representations in a large range of problems is an important piece of future work, but is not the aim of this thesis. The purpose of the experiments here is to demonstrate with a real example some important points about representations. They aim to investigate the following questions.

- Given a variable representation $R = \langle \overline{V}, f \rangle$, does randomising f, such that the representation is still valid, make any real difference on problems where R had a completely perfect GAC implementation?
- Given a variable representation $R = \langle \overline{V}, f \rangle$, does randomising f, such that the representation is still valid, make any real difference on problems with random constraints?
- Given many representations, is it possible to add them all to a model and use channelling constraints and get as small a search as any of the representations in isolation?
- Using multiple representations and channelling can clearly increase search time. Can it increase search size?

Having read this thesis, the answers to these questions may be unsurprising, but the answers to these questions have surprised a number of senior constraint practitioners.

Investigating these questions involves first defining exactly what a random representation is. While in theory it would be possible to generate arbitrary random representations, producing a solver which can implement an arbitrary representation would be difficult. Variable representations on the other hand are specified only as an surjective function from a list of domains to a single domain. Furthermore, variable representations can be used to transform one CSP into another. Therefore implementing any variable representation requires only a solver which can handle extensional constraints.

One obvious advantage of using a variable representation with a compact implicit form is that this representation can be contained in a small amount of space. Researchers have designed compact and efficient propagation algorithms for propagating common constraints on the representations considered in this thesis, For a random representation, such implementations will not exist and in general there may not exist an efficient implementation. In the case where only the size of searches are being compared and the level of propagation on all constraints achieved on all constraints is the same however, this advantage is removed. In this case, do carefully crafted representations provide any gain over a random representation?

Random representations in this section are generated using the algorithm in Definition 7.1. Example 7.1 gives a concrete example of a random representation and demonstrates how they can lead to a loss of propagation.

Definition 7.1. A *randomised* copy of a variable representation $\langle \overline{V}, f\rangle$ of the domain D is a variable representation $\langle \overline{V}, f'\rangle$ of D where f' is any partial surjective function.

Example 7.1. *The occurrence representation for subsets of* $\{1,2,3\}$ *is defined by* $\langle\ \langle v_1, v_2, v_3\rangle, f\rangle$ *where* f *is defined by the table:*

$$\begin{array}{ll} f(0,0,0) = \{\} & f(0,0,1) = \{3\} \\ f(0,1,0) = \{2\} & f(0,1,1) = \{2,3\} \\ f(1,0,0) = \{1\} & f(1,0,1) = \{1,3\} \\ f(1,1,0) = \{1,2\} & f(1,1,1) = \{1,2,3\} \end{array}$$

One randomised copy of $\langle \overline{V}, f\rangle$ *is* $\langle \overline{V}, f'\rangle$, *where* f' *is defined by the following function:*

$$\begin{array}{ll} f'(0,0,0) = \{2\} & f'(0,0,1) = \{1,2\} \\ f'(0,1,0) = \{2,3\} & f'(0,1,1) = \{1,2,3\} \\ f'(1,0,0) = \{1\} & f'(1,0,1) = \{\} \\ f'(1,1,0) = \{1,3\} & f'(1,1,1) = \{3\} \end{array}$$

On this random representation, consider propagating the constraints $1 \in S$ *and* $1 \notin S$. *Each assignment to each of the* v_i *can be extended to an assignment which represents a set which contains* 1*, and one which represents a set not containing* 1*. Therefore while performing GAC on both these constraints in the standard occurrence representation would lead to domain wipe-out, in this random model it would not.*

In the experiments in this chapter all representations will be treated equally, with the same level of propagation performed on the constraints for each representation regardless of whether the representation was randomly generated or one of the previously discussed representations. Therefore any differences between sizes of the search trees will be based only on the representations. These experiments continue the theme of the thesis so far by not reporting the time taken in any problem. The reason for this is that the propagation algorithms for random representations use very slow general algorithms. For any particular random representation there may well exist a much faster algorithm, comparable in speed to those for the standard *occurrence* representation, but this was not investigated.

7.1 BIBDs

The previous discussions of dominating representations and representations with completely perfect GAC implementations of some constraints showed that the *occurrence* representation satisfies a number of theoretical properties that makes it a good representation. In the vast majority of cases, random representations will not satisfy any of these properties. Therefore it would seem reasonable that, for example on problems involving sets and common set constraints, such as intersection, subset and size then the *occurrence* representation should outperform its random counterparts.

A very commonly used benchmark in constraint programming is the Balanced Incomplete Block Design (BIBD) problem (Definition 4.2 on page 49). There are a number of possible ways of defining this problem, at different levels of abstraction. The problem is expressed in Definiton 7.2 as it will be implemented in this section.

Definition 7.2. A $\langle v, b, r, k, \lambda \rangle$ BIBD requires, finding an array of b subsets of $\{1, \ldots, v\}$ of size k, where each element of $\{1, \ldots, v\}$ occurs in r sets and the intersection of each pair of sets has size λ.

7.1.1 Implementation Details

Representing the constraints of the BIBD intensionally using the *occurrence* representation is very simple. The major problem with expressing the constraints extensionally is the constraints which constrain the number of sets which must contain each value between $\{1, 2, \ldots, n\}$. In the case where the sets are represented using a random representation, every variable in a set may be required to find if it contains a given value. Therefore this constraint will involve every variable in the BIBD. Theorem 6.6 will be used to keep the arity of constraints to a manageable size.

Theorem 6.6 on page 103 shows that given a constraint which can be expressed as:

$$g(f_1(\overline{V_1}), f_2(\overline{V_2}), \ldots, f_n(\overline{V_n}))$$

for functions g and f_i and disjoint arrays of variables $\overline{V_i}$, the propagation achieved on this constraint is the same as the propagation achieved on each of the constraints in the flattening:

$$\{g(x_1, x_2, \ldots, x_n), x_1 = f_1(\overline{V_1}), \ldots, x_n = f_n(\overline{V_n})\}$$

generated by introducing new variables x_i. In particular, if the new variables x_i are left until the end of search ordering, then the search tree is unaffected by adding these new variables and splitting the constraint. For each value $j \in \{1, \ldots, n\}$, the constraint that j must occur in exactly r sets would, on random representations, involve every variable of the BIBD and therefore would quickly become impossible to implement. This simplification allows us to instead use constraints of the form $b_{i,j} \iff j \in S_i$ for new booleans $b_{i,j}$, set variable S_i and constaint j, and then impose $\forall j. \sum_i b_{i,j} = r$.

There are two minor points which must be recalled about Theorem 6.6. Firstly it assumes that no two of the $\overline{V_i}$ have a variable in common. Secondly if the same expression $b_i = f_i(\overline{V_i})$ is introduced while flattening two separate constraints, then a new b_i should be introduced for each constraint. Reusing the same b_i is correct but will in general decrease the size of search. While in general this would be a good thing, here Theorem 6.6 is being used only as an implementation detail to

	Average	Std. Dev.	Min	Max	% Best
occurrence	977.87	237.3	616	1795	84%
randomised copy	3140.03	691.82	1658	4871	0%
	1291.24	220.05	898	1912	8%
	2957.44	519.80	1542	4145	0%
	2480.00	412.29	1609	3590	0%
	1815.32	335.50	1134	2680	0%
	1965.55	356.78	1004	2782	0%
	1279.84	197.40	817	1801	8%
	2702.22	539.52	1444	3772	0%
	2170.29	419.80	1274	3231	0%

Figure 7.1: The *occurrence* representation, and 9 randomised copies of the *occurrence* representation, tested on 100 random variable orderings of the $(5, 5, 2, 2, 1)$ BIBD.

allow the more efficient implementation of extensional constraints of large arity.

7.1.2 Comparing the occurrence and random representations

This first experiment aims to compare solving BIBDs with the *occurrence* and nine randomised copies of the *occurrence* representation. If the mapping from a representation to the original domain was unimportant, we would expect the random representations to perform as well as the original *occurrence* representation.

The results of using both the *occurrence* representation and nine randomly generated representations for the $(5, 5, 2, 2, 1)$ BIBD is presented in Figure 7.1. In each case, one hundred experiments were run, each using a random variable ordering, proving that these instances of the BIBD problem have no solution. It can be clearly seen that the original occurrence model performs much better. In both cases it produces both the smallest instance, the lowest average and the smallest upper limit. For the $(5, 5, 2, 2, 1)$ BIBD it is the best 84% of the time. This shows the result, unsurprising based on the previous content of this thesis, that representations cannot simply be compared by the number of representational states they allow, but the exact representation is of vital importance.

Consider the concrete random representation given in Example 7.1 and the example that it is not possible to propagate either $1 \in S$ or $1 \notin S$. This pattern can occur frequently, meaning until almost every variable is assigned, no propagation will occur. This leads to very little ability for the constraints to pass information, as this only occurs through propagation to reduce the domains of variables.

This result relates back directly to the idea of representations with completely perfect GAC implementations of some constraints. A good representation must be able to represent the results of the deductions which propagators make, so that the results of these deductions can pass between constraints. When propagators are unable to place the results of deductions into the domains of the variables, then the size of search increases substantially.

7.2 Channelled Representations

A number of papers have shown how channelling can help reduce the size of search, as discussed in Section 3.1. Channelling is similar to joining representations, as described in Definition 4.12 on page 60. An obvious question therefore is if channelling representations can be used to improve the performance of a group of poorly performing representations.

Channelling is not a silver-bullet which can be used to always get the best features of multiple models, even when ignoring the fact that adding multiple models will increase the time taken at each node. Channelling or joining multiple models should be used with thought and caution, as it can in some cases cause the size of search to increase and others the reduction in search to be much smaller than expected. The major cause of this is *model drift*, which is demonstrated in Example 7.2.

Example 7.2. *Consider representing an integer in range* $[1, 100]$ *by two variable representations. The first, denoted* $\langle \overline{V_B}, f_B \rangle$*, represents the number in binary, while the second, denoted* $\langle \overline{V_T}, f_T \rangle$*, represents the integer in ternary. This means* $V_B[0] = 0$ *means the integer is divisible by 2, and* $V_T[0] = 0$ *means the integer is divisible by 3. Consider the case where both of these representations are being used, and a channelling constraint is placed between them.*

If the first branch made during search is $V_B[0] = 0$*. This constrains the variable to be even. This does not allow any propagation to occur to the array* $\overline{V_T}$*, as for every assignment to every variable, there is an assignment to the other variables where the result is even. Similarly, assigning* $V_T[0] = 0$ *causes no propagation to* $\overline{V_B}$*.*

Example 7.2 shows how propagating assignments can reduce the variables in different models without propagating this change between the models. In the most extreme cases, almost all variables can be assigned in both models without any propagation occurring.

There are two standard methods of using multiple channelled representations in the same CSP. The branching strategy may either be free to choose a variable from any of the representations to branch on, or a single representation can be chosen to be branched on and the others used only to increase the level of propagation and reduce search.

Branching on only one model has the advantage that in many cases it can be proven that this will not increase the size of search as long as a static branching strategy is used, as it can only lead to more propagation. However branching on multiple models can lead to smaller searches, for example as discussed by [19].

This can lead to increased search in two different cases. Firstly consider that the branching method can choose freely between the two sets of variables and alternate between them. The problem can degenerate into a case where two copies of the problem are being almost independently solved at the same time, which would massively increase the search space. Even if the sub-domains of both problems contain a solution, they must contain the same solution for the whole CSP to be solvable.

The second problem, independent of the variable ordering used, is if the modelling is non-redundant, that is that some constraints are imposed only on one of the models, and some only on the other. In this case if only one model is branched on, the other will have almost no propagation until almost all variables are assigned, at which point the other constraints will be checked, leading to a result where some constraints are effectively not checked until all variables are assigned. This can clearly massively increase search size.

This leads to the question of how to reduce this problem. One of the most obvious methods and the one which has stopped this problem being a major problem in most previous work on channelling is to ensure that model drift does not occur. In particular, when the alternative models are literal equivalent (Definition 5.9 on page 85), and the channelling constraints achieve GAC, then no model drift can occur as the different representations will at all points represent exactly the same set of assignments. However, in this case there is no gain in representational power by channelling between two such representations.

Another option is to only branch on one of the channelled representations. In this case as the multiple representations joined together dominate any one of the representations in isolation, the search space cannot be bigger assuming a static search ordering. However any reduction in the size of the search over a single representation may be small.

7.2.1 Channelling Experiments

To demonstrate the possible problems which can arise during channelling, a number of experiments were performed using joined representations. These consider branching on both of the original representations and on only one of the original representations.

7.2.1.1 Branching on One Representation

These experiments compare solving BIBD with a single representation with adding a second channelled model while keeping the branching on the first model. Once all variables in the first representation are assigned, the ones in the second will become assigned by propagation. With a static variable ordering this cannot lead to an increase in search size. Hence, the experiments only aim to show how much the size of search is reduced. This experiment is important, because it shows how effective adding extra redundant models is to improving search.

Three sets of experiments were performed. Each model was tested with one hundred different randomly generated variable orderings. In each case only the first model is branched on and the ordering heuristic kept the same when a second model was added and channelled to. The aim of these experiments is to demonstrate that in general channelling one model of the BIBD to a second model of a problem while keeping the branching only on the first model can lead to almost no improvement.

First, the *occurrence* representation alone was compared to the *occurrence* representation channelled together with 10 different randomised copies of the *occurrence* representation to prove there are no solutions to the $(5, 5, 2, 2, 1)$ BIBD. On average this took around 977 search nodes. None of the 1,000 experiments improved the size of search by a single node.

Secondly, a randomised copy of the *occurrence* representation was compared to the same randomised copy joined with the standard *occurrence* representation. Once again the branching only operated on the first model. The average number of search nodes reduced from 3140 to 3020, a 3.8% improvement. In every instance there was some improvement, if only a handful of nodes, but in no experiment

	Average	Std. Dev.	Min	Max	% Best
occurrence	977.87	237.3	616	1795	48%
Two copies of occurrence	969.6	223.99	635	1664	52%
occurrence + randomised copy	3888.15	1647.22	1082	8027	0%
	1902.58	648.55	879	4172	0%
	3514.4	1542.32	1089	8770	0%
	3436.6	1532.24	1164	8865	0%
	2395.72	961.01	978	6091	0%
	2624.19	1032.62	1027	6589	0%
	1718.91	497.49	880	3131	0%
	4176.16	1995.13	1279	11294	0%
	2844.41	1023.59	873	5143	0%

Figure 7.2: Compare the *occurrence* representation, channelling the *occurrence* rep with itself, and channelling the *occurrence* rep with 9 random representations. Tested on 100 random variable orderings of the $(5, 5, 2, 2, 1)$ BIBD.

was the gain greater than 9%. These times are still very poor in comparison to the standard *occurrence* representation alone, where the average number of search nodes was only 977.

The final experiment involved a random copy of the *occurrence* representation joined to a different random copy of the *occurrence* representation. One random representation was chosen and then channelled with 10 different other random models. Of these 10, 6 produced no gain in any of the 100 experiments performed, 3 produced a gain of less than 1%, and one produced a gain of around 4%.

What is most interesting about these experiments, in particular the comparison of attaching the *occurrence* representation to a random representation, is the very small gain in performance. This shows that a very poor representation cannot be improved by attaching a much better performing representation if the better representation is not considered in the branching.

7.2.1.2 Branching on Multiple Representations

These experiments are similar to those in subsection 7.2.1.1, except here once the representations are joined the branching strategy branches over all variables.

Figure 7.2 shows results from channelling two models together, one of which is the standard *occurrence* representation and one of which is random. For comparison, the first two results are simply the standard *occurrence* representation by itself and two copies of the standard *occurrence* representation channelled to-

	Average	Std. Dev.	Min	Max	% Best
occurrence	8412.7	2494.32	3687	16492	49%
Two copies of occurrence	8485.98	2486.72	3697	16489	50%
occurrence + randomised copy	112764.15	81294.2	18972	480484	0%
	110814.23	72667.91	21067	448059	0%
	75751.99	46924.48	17014	279648	0%
	72993.3	42767.6	18935	221033	0%
	93568.35	58737.88	20165	325405	0%
	98183.81	64743.24	20672	381414	0%
	87401.62	53234.48	17375	303853	0%
	69614.75	41266.01	19275	276068	0%
	52511.66	28417.99	16189	149921	0%

Figure 7.3: Compare the *occurrence* representation, channelling the *occurrence* rep with itself, and channelling the *occurrence* rep with 9 random representations. Tested on 100 random variable orderings of the $(6, 6, 2, 2, 1)$ BIBD.

gether. This second result should be effectively identical to the first except for some minor variations caused by a different variable ordering. This is because whenever one variable in one of the two models has a value removed from its domain, the same value will be removed from the domain of the same variable in the channelled representation. This is backed up by the results in Figure 7.2

Without exception, channelling the original and a random representations performs very poorly. In particular, in no case does it outperform either the original representation, or the random representations (which are the same as those given in Figure 7.1) in isolation. The $(6, 6, 2, 2, 1)$ BIBD experiments, shown in Figure 7.3 shows these results even more strongly on a larger problem, with the performance of the models involving a random representation even poorer.

These BIBD experiments, while limited in scope, have shown that (as might be hoped) the *occurrence* representation does perform better in practice than a random representation. Also they show that in general, channelling is not a magic bullet and provides almost no help in combining a number which can be used to improve performance, by combining a number of random representations.

7.3 Random Problems

The previous section showed how the *occurrence* representation outperforms randomised copies of the *occurrence* representations on BIBDs, a problem which contains many common set constraints. This raises the question of whether the

	Average	Std. Dev.	Min	Max	% Best
Original	260842.31	54209.18	132424	423110	26.5%
Random 1	260625.90	54444.54	132491	426272	27.5%
Random 2	261407.72	54348.02	127135	414420	23.5%
Random 3	261173.01	54423.89	125776	419634	22.5%

Figure 7.4: Comparison of number of search nodes on random problems of the *occurrence* model and three random variants. All results to 2 decimal places.

	Occurrence	Random 1	Random 2	Random 3
Average Nodes alone	260842	260626	261408	261173
Av. gain with Occurrence	0	1129	1884	1906
Av. gain with Random 1	1414	0	2188	2488
Av. gain with Random 2	4329	1400	0	2235
Av. gain with Random 3	2115	2311	2017	0

Figure 7.5: Comparing four models of a set on a random problem with attaching an auxiliary model without changing branching strategy.

occurrence representation would outperform a random copy for all problems.

One obvious way to compare the *occurrence* representation and random copies to see if they perform the same is on random problems. As these constraints have no particular structure, there should be no reason for them to perform better on the *occurrence* representation, unless it has some structure which allows it to perform better for all constraints. We expect that the *occurrence* representation and random copies are going to perform identically here.

In each problem in this section, a problem is defined over 5 set variables of domain $\{1, 2, 3, 4, 5\}$. A constraint is placed between each pair of variables, and each tuple is included in each constraint with probability $\frac{1}{2}$. In the case of problems which have solutions, the entire search space is explored. For each data point, 200 random problems are generated and tested. In each experiment, the same 200 problems and branching strategies are used.

Figure 7.4 results show that the original *occurrence* representation is no better or worse than any of the random representations, with the average best and worst search trees about the same, and the original *occurrence* model performing best on 26.5% of problems, showing the 4 models are effectively as good as each other.

Figure 7.5 shows an experiment where the standard *occurrence* representation and three random variants were used to solve 200 randomly generated problems. On the same problems with the same branching order, each model was channelled

	Occurrence	Random 1	Random 2	Random 3
Occurrence	261000	439149	438588	430980
Random 1	-	261788	440850	439735
Random 2	-	-	262047	438042
Random 3	-	-	-	262045

Figure 7.6: Trying all pairs of four models of a set on a random problem, including channelling each model to itself.

to each of the other models in turn. This could only decrease the size of the search tree. The values in the diagonal are 0 in all cases, as here the model is channelled to a copy of itself, proving no gain. These results show two important facts. Firstly there is no advantage to either starting with the standard *occurrence* representation or channelling to it. Secondly adding a second model produces a gain of less than 1%.

The final set of experiments on random problems, given in Figure 7.6, tries channelling two models together and branching on both sets of variables. Unsurprisingly, once again the two main features are that the standard occurrence model is no better than the random models and the channelled models perform very poorly except when the two modells being channelled together are identical.

These results show that on random problems, there is no difference between the performance of the *occurrence* representation and a randomised copy. They also show that using multiple representations produces an extremely small gain at best, and a large decrease in performance at worst. These channelling results are the same as those achieved for BIBDs.

7.4 Conclusion

This chapter has presented a number of experiments which put this thesis in context. The most important set of results in this chapter are those involving channelling. These have shown that using multiple representations and channelling can result in much worse performance. Further, given a number of models without knowledge of which is better is not useful in practice. Branching on all the models can be very poor and when when branching on just one a poor choice of model to branch on cannot be improved by channelling to good models. A more complete investigation of exactly when channelling is successful is an important future research area but outside the scope of this thesis.

This chapter also provided some evidence toward the general claims of this thesis, namely that representations cannot be considered solely in terms of the number of representative states they represent. It has shown how on the common set constraints, the *occurrence* model outperforms a random alternative. This makes sense, as it has a completely perfect GAC implementation of many of the set constraints used in the BIBD problem. The random representations on the other hand perform very poorly. On random problems however, the *occurrence* representation does not better than random variants. This shows the unsurprising result that when dominance cannot be proved between representations, it is necessary to consider the problem in which they are to be used.

Chapter 8

Breaking Symmetry in Representations

8.1 Introduction

As discussed in Section 2.4, symmetry breaking is an active and important research area in CP. A large number of experiments have shown how breaking symmetry can massively decrease the size of search trees and therefore the time taken to solve problems.

However, effectively making use of symmetry in CP is not trivial and there are a number of important issuess which must be addressed. Firstly, there are different possible definitions of symmetry, which each have their own advantages and disadvantages. Secondly, given a CSP and a symmetry definition, the symmetry of a CSP must either be given by the user, or more usefully detected automatically. Finally, an algorithm must be designed to use the symmetries found to usefully reduce the search size and hopefully also the search time.

Even with the current best-known methods, symmetry breaking is not a silver bullet. For example, the experiments performed by Petrie [62] on Balanced Incomplete Block Designs show examples where symmetry breaking slows the solving process from thousands of nodes per second to 30 seconds per node. Part of the reason is that in that the symmetry breaking method used, GAP-SBDD, introduced in Section 2.4.4, involves solving an NP-complete problem at every node of search. In those experiments however, the reduction in search space is still sufficiently great that problem can be solved which can not be possible without symmetry breaking.

8.2 Traditional CSP Symmetry

There is no one definition of symmetry which has gained acceptance as the standard in constraint satisfaction. The general definition of a symmetry, previously given in Definition 2.21, is repeated here in Definition 8.1. Using this basic idea, defining symmetries of a CSP involves chosing the properties which must be preserved and the set which is permuted. At first both of these seem obvious, but in practice these must be defined carefully, and what may at first appear to be equivalent definitions end up differing in subtle ways.

Definition 8.1. A bijective function f is a symmetry with respect to property p of a set S if $\forall s \in S.p(s) \iff p(f(s))$.

One problem which shows the subtle problems involved with defining the symmetries of a CSP is the n-queens problem.

Example 8.1. *The n-queens problem requires placing n queens on an $n \times n$ chessboard such that no two queens can attack each other. This means no two queens can be on the same row, column or diagonal. Diagram 8.1 shows two symmetric solutions to the 8-queens problem.*

There are three commonly used models for this problem. The first, known as the Boolean *model, uses $n \times n$ array M of Boolean variables, where $M[i, j]$ is* True *if there is a queen at location $\langle i, j\rangle$. The second model, known as the* row *model, has a variable of domain {1,2,...,n} for each row, representing where the queen is in that row. The* column *model is defined similarly.*

Intuitively, the n-queens problem has the same symmetries as a chessboard, which are rotations by 90, 180 and 270 degrees, reflections vertically, horizontally and along the two diagonals and combinations of these.All combinations of these symmetries form one of the symmetries already listed, or the identity symmetry which does not move the chess board, so the symmetries form a group of size 8.

An obvious definition for symmetries of a CSP is a permutation of the set of assignments which maps solutions to solutions and non-solutions to non-solutions. This has two major problems. The first is that it is too restrictive. Consider the row model of the n-queens problem and a 90 degree rotation. The assignment where every queen is placed in the first column is mapped under 90 degree rotation to having all queens in a single row. This means an assignment is mapped to a non-assignment, as the first variable takes all values and the other variables take

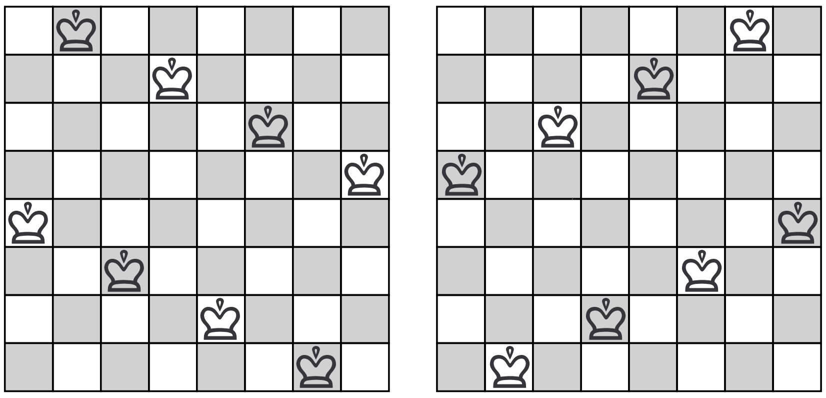

Figure 8.1: A solution to the 8-queens problem, and its symmetric image under horizontal flip.

no value. Therefore while any sensible definition of the symmetries of a CSP must map solutions to solutions, it should not be required that non-solutions map to non-solutions.

The second, less serious, problem with defining the symmetry group of a CSP as a permutation of the set of assignments is that this allows too much. Given n variables of domain size d there are d^n assignments, and therefore $(d^n)!$ permutations. One standard result of group theory is that it is possible to represent any group over n variables with a set of generators of size at most n. However, this still means specifying the symmetry group of a CSP can require d^n permutations. Furthermore, the most general symmetry group of any CSP will simply allow any pair of solutions to be permuted, and any pair of non-solutions to be permuted. This is too large and too general to be of practical use.

The most common definition of symmetry used in CSPs has been instead to define symmetries as a permutation of the set of literals of the variables. This has many advantages:

- This definition in practice appears to capture most of the symmetries that CP practitioners observe.
- Expressing the group as a list of generators takes only $O(n^2d^2)$ for a CSP with n variables of domain size d.
- As the set of literals is often stored in the solver during search these groups map naturally to the internal state of the solver, operations on these groups map naturally to the state of search.

There are symmetries this definition does not encapsulate, most notably conditional symmetries [37]. However such symmetries are at present poorly studied and there is no general practical algorithm to make use of them in CP. From now on unless stated otherwise, all references to the symmetry group of a CSP can be assumed to refer to a group defined over the set of literals of the CSP.

There is still one major problem with the definition of symmetry given thus far, which is that finding the symmetry group of a CSP appears to require knowing the complete set of solutions. In fact, as Corollary 8.2 of Theorem 8.1 shows, given the symmetry group of a CSP it is trivial to check if it has any solutions.

Theorem 8.1. *Given a CSP P with at least two variables, the symmetry group of the solutions of P is the complete group over its literals if and only if P has no solutions.*

Proof. The definition of a solution symmetry requires that it maps solutions to solutions. Therefore in a CSP with no solutions, any automorphism of the literals is a symmetry.

In a CSP with at least one solution and at least two variables, take any solution and consider two assignments to two distinct variables in that solution. As a symmetry must map solutions to solutions, these two literals can never be mapped to the same variable by any symmetry. □

Corollary 8.2. *Given a CSP, knowing if the symmetry group is complete permits checking if the CSP has a solution by checking if only a single assignment is a solution.*

Proof. Theorem 8.1 shows the symmetry group is complete if and only if the CSP has no solution for CSPs with at least two variables. In the degenerate case of a CSP with only one variable, the symmetry group will be the union of the complete symmetry group over the set of solutions and the complete symmetry group over the set of non-solutions, so the symmetry group will be complete if the problem has either no solutions or every assignment is a solution. Which of these cases holds could be checked by checking only a single assignment. □

While finding all symmetries of a CSP may be difficult, it is often possible to find a large number of symmetries of a CSP. In many cases, symmetries are found by inspection of the problem by a CSP practitioner and with practice it becomes easier to identify many common classes of symmetry. Automating symmetry detection would remove the need for expert practitioners to identify symmetries.

One useful theoretical tool for this is the microstructure graph, given in Definition 8.2.

Definition 8.2. Given a CSP P, the microstructure complement graph contains a single node for each assignment to every variable in P and a hyper-edge between the assignments disallowed by each tuple in each constraint. It also contains a hyper-edge between each pair of assignments to each variable, to represent no variable may take more than one assignment.

Figure 8.2 shows the microstructure complement graph of a simple CSP. Theorem 8.3 shows that any symmetries of the microstructure graph are also symmetries of the CSP.

Theorem 8.3. *Any permutation of the vertices of the microstructure complement graph of a CSP which is a symmetry of that graph also maps solutions of the CSP to solutions.*

Proof. Solutions are sets of vertices in the microstructure graph which form an independent set, that is there is no hyper-edge which is made up solely of vertices of the solution. Applying any symmetry of the microstructure graph to such a set cannot result in it mapping to a new set of vertices which do have an hyper-edge made only from members of the solution. □

The converse is not true, for example Theorem 8.1 showed that any CSP with no solutions has the complete solution symmetry group, but the microstructure of most CSPs with no solutions will not have all symmetries. Detecting the symmetries of the graph is a problem which does not have a known complexity and it has not been proved if it is polynomial, NP-complete or lies between these two. However in practice very large graphs can be handled easily. With the microstructure graph, it is possible to give the two commonly used definitions of CSP symmetry.

Definition 8.3. A *solution symmetry* of a CSP is one which maps solutions to solutions. A *constraint symmetry* of a CSP is one which maps the microstructure graph to itself.

One problem with this system of detecting symmetry is that it requires turning any intensional constraint into an extensional form. As always, for large arity constraints this can be difficult, or not even practical. Work in automatically

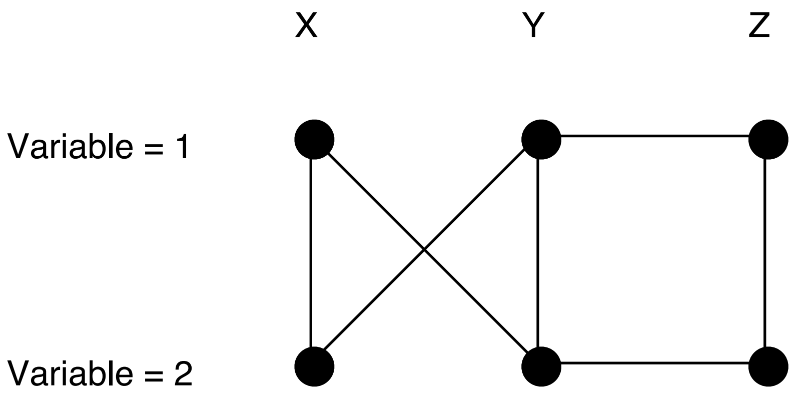

Figure 8.2: The microstructure complement graph of $X = Y, Y \neq Z$ for variables $X, Y, Z \in \{1, 2\}$

detecting symmetries [57,65] shows how to build graphs which encode intensional constraints in a more compact form. This can lead to a slightly decreased number of detected symmetries, but ensures the size of the generated graph is polynomially bounded by the size of the original intensional CSP. In practice, the number of detected symmetries is almost always identical.

8.3 Recursive Symmetry Breaking

As discussed in Section 4.3, one of the major advantages of variable representations is they can be applied recursively. This is often performed during the refinement of a high-level specification of a problem into the CSP which will actually be implemented. The symmetries generated from these repeated applications of representations could all be combined and then dealt with in one step. However keeping the nested structure of the symmetries allows more efficient methods of dealing with the symmetries.

In the case of static symmetry breaking, it is simple to break symmetries which have been formed by repeated applications of variable representations, as shown in Theorem 8.4. This allows symmetries which are nested inside each other to be broken independently, avoiding concerns about how the symmetry breaking methods used interact.

Theorem 8.4. *Consider a variable representation $\langle \overline{V}, f \rangle$ of a domain D with valid symmetry breaking constraint C and for each element of $\overline{V}$ a representation $\langle \overline{V_i}, f_i \rangle$ with valid symmetry breaking constraint C_i. Then on the representation made by joining $\langle \overline{V}, f \rangle$ and the representations $\langle \overline{V_i}, f_i \rangle$, the image of C under* *the representations $\langle\overline{V_i}, f_i\rangle$ and the constraints C_i is a valid symmetry breaking constraint. Further, if C and the C_i were each complete, then the new constraint is also complete over the whole representation.*

Proof. Call the variables after all the representations are joined an array of arrays $\overline{W}$, where the i^{th} element of $\overline{W}$ is the variables of $\overline{V_i}$. Then for any assignment d to D there must be at least one assignment $\overline{V}$ which represents it and also satisfies C. In the case where C is complete, there will be exactly one such assignment.

Given an assignment a to $V[i]$, there must be an assignment to $\overline{V_i}$ which represents a and satisfies C_i. If C_i is complete, there must be exactly one such solution. Therefore the both C and C_i together are consistent, and if they are all complete then the join is complete. □

One of the major advantages of breaking each layer of a recursive symmetry separately rather than combining the various symmetry groups into one is that the number of constraints required is greatly reduced. For example the *Crawford* [16] method of symmetry breaking requires $n - 1$ constraints for a group of size n. In the case of a array $\overline{V}$ where the array has index symmetry group G_A, and each element has a variable symmetry group G_V, then the size of the combined group and there the number of required constraints is $|G_A - 1| * |G_V - 1|^n$. In the case where the symmetries on $\overline{V}$ and its elements are broken separately only $|G_A - 1| + |G_V - 1| * n$ constraints are required.

Petrie [62] showed using different dynamic symmetry breaking methods on different symmetry groups in the same problem at the same time in general does not work, as dynamic symmetry breaking methods involve replacing the standard search procedure with an altered, symmetry aware one. How to combine dynamic symmetry breaking methods shall therefore not be considered. This still leaves the case of applying a combination of dynamic and static symmetry breaking to a recursive symmetry.

Given a recursive symmetry, after applying either a change of representation or adding constraints to break the symmetry of the lower level, the symmetry of the upper level still exists. Hence, in this case it is valid to apply a dynamic symmetry breaking method. This is often done in practice, where it is obviously valid to apply dynamic symmetry breaking to an index symmetry of $\overline{V}$ and static symmetry to its elements, as after applying the static symmetry breaking the resulting CSP still has the same index symmetry as before. The reverse is not true, as Example 8.2 shows.

Example 8.2. *Consider an array of arrays of variables $\overline{V}$, where $\overline{V}$ has two elements, each an array of 2 elements, where both V and its elements have complete index symmetry. If $V[a,b]$ refers to the b^{th} element of the a^{th} element of $\overline{V}$, then a valid symmetry breaking constraint for the index symmetry of V is $V[1,1]V[1,2] \leq_{lex} V[2,1]V[2,2]$.*

Consider the case where $V[1,1] = 1, V[1,2] = 2, V[2,1] = 2, V[2,2] = 1$ and its symmetric equivalents are solutions. If the first symmetric assignment to this which arises during search is $V[1,1] = 2, V[1,2] = 1, V[2,1] = 1, V[2,2] = 2$, then this will be disallowed as a solution by the static symmetry breaking. However, this dynamic symmetry breaking will now forbid any assignment which is a symmetric image of this, which will include all the solutions.

8.4 Problem Specific Symmetry Breaking

There have been a number of general complete and incomplete static symmetry breaking methods, such as the lexicographic [26] and multiset [29] orderings, which allow many symmetry groups to be broken. For certain commonly occurring symmetry groups, methods such as the "double lex" constraints for row and column symmetries, provide faster group-specific symmetry breaking methods. However, there has been relatively little research into problem-specific symmetry breaking. Example 8.3 demonstrates one simple example of how a clever choice of symmetry breaking constraint can theoretically decrease search size.

Example 8.3. *Consider a CSP with a fixed size set variable S, whose domain is subsets of $\{1, \dots, m\}$ of size n and the constraint "S must include an even number". If S were represented by the explicit representation with explicit array $\overline{E}$, this constraint is implemented as $\exists \mathbf{s} \in \{\mathbf{1}, \dots, \mathbf{n}\}.\ \mathbf{E}[\mathbf{s}]$* ***is even****.*

As $\overline{E}$ has symmetry, it can be broken. One way of doing so would be to alter the representation so that the elements of E must be ordered. This would indeed break all the symmetry but leave the constraint identical. Consider instead ordering E such that first come all even values, then all odd values, and within these ranges the integers are ordered as normal. This can be considered identical to normal lexicographic symmetry breaking except using a different ordering of the integers, where all even numbers are smaller than all odd numbers. In this case, if any element of E is even the first is, so the constraint becomes $\mathbf{E}[\mathbf{1}]$ ***is even****, a much simpler constraint which will propagate straight away.*

The above example shows the underlying idea behind this section. By choosing symmetry breaking constraints carefully it is often possible to allow constraints to be more efficiently expressed. Usually the purpose of symmetry breaking is to avoid searching redundant parts of the search space. However, symmetry can also reduce propagation. While it is possible to propagate while leaving at least one symmetric image of each solution, as in Example 8.3, it is not possible to do so without removing some solutions. Therefore it is worth investing in symmetry breaking methods which as well as removing the symmetry of the problem, leave the existing solutions so that the constraints propagate better. In particular the best case is if after applying symmetry breaking, some constraints gain a completely perfect GAC implementation.

One very common pattern in CSPs with set and multiset variables, either in the constraints of a problem or in implied constraints which can be found, is to require one of more elements of the (multi)set to satisfy a certain set of properties. One pattern which can be used to represent many examples, of the type given in Example 8.3, is given in Definition 8.4.

Definition 8.4. Consider a (multi)set variable X whose domain is subsets of some fixed set S and a function f from S to $\mathbb{R}$, where X is represented under the *explicit* representation with element array E. The **f-lexicographic constraints** are $(\mathbf{f}(\mathbf{E}[\mathbf{i}]) \leq \mathbf{f}(\mathbf{E}[\mathbf{i}+\mathbf{1}])) \vee \mathbf{E}[\mathbf{i}+\mathbf{1}] = \emptyset$, where $\emptyset$ represents $E[i]$ not being in the (multi)set. If the set is fixed size, the constraint is only $\mathbf{f}(\mathbf{E}[\mathbf{i}]) \leq \mathbf{f}(\mathbf{E}[\mathbf{i}+\mathbf{1}])$.

Lemma 8.5 shows that the constraints given in Definition 8.4 are valid symmetry breaking constraints, and in what case they are complete. The major use of these constraints is given in Theorem 8.6, where they give constraints of the form $\exists \mathbf{s} \in \mathbf{S}.\ \mathbf{f}(\mathbf{s}) \leq \mathbf{n}$ completely perfect GAC implementations on the *explicit* representation.

Lemma 8.5. *Given a (multi)set variable X whose domain is subsets of some fixed set S and some function f from S to $\mathbb{R}$, the f-lexicographic symmetry breaking constraints, as defined in Definition 8.4, are valid symmetry breaking constraints for the explicit representation of X. f is complete if and only if it is injective.*

Proof. Consider X is represented by the *explicit* representation with check array $\overline{E}$. Given any assignment $\overline{e}$ to $\overline{E}$, first sort it so all occurrences of $\emptyset$ (the value which denotes that position is not used) are at the end. Then sort all other values by their image under f. This assignment will satisfy the f-lexicographic symmetry breaking constraints.

If f is injective, then this sorting must always be unique. If f is not injective, choose an assignment to X which includes two elements which have the same image under f. Then given any assignment which satisfies the constraints, permuting these two elements will give another assignment which does. □

The next theorem provides the main result of this section, showing how the correct choice of symmetry breaking can make some constraints perfect on the explicit representation.

Theorem 8.6. *Given a (multi)set variable X whose domain contains all sub-(multi)sets of some (multi)set S and a function f from S to $\mathbb{R}$, the explicit representation with symmetry broken by the f-lexicographic ordering has a totally perfect GAC implementation of the constraint* ***at least*** *m* ***elements of*** *X* ***satisfy*** $\mathbf{f(s) \leq n}$ *for any m.*

Proof. On the symmetry broken *explicit* representation with element array $\overline{E}$, this constraint is equivalent to $\forall \mathbf{i} \in \{\mathbf{1}, \ldots, \mathbf{m}\}.\ \mathbf{f(E[i]) \leq n}$, which is logically equivalent to the split into the a set of constraints, each of the type $\mathbf{f(E[i]) \leq n}$, which satisfies the condition required for a constraint to have a completely perfect GAC implementation. □

An obvious simple extension of Theorem 8.6 would be to allow both a constraint which required some element of a set to satisfy $\mathbf{f(x) \leq n}$, and another to satisfy $\mathbf{f(x) \geq m}$. These constraints can both be made perfect at the same time, one placed at one end of the element array and the other at the other. In the case where the set is of fixed size, this follows simply from symmetry. In the general case it becomes trickier, as the right-most element of the element array may not represent a value in the set at all, and in this case it is necessary to instead impose the more complex, and not completely perfect GAC, constraint which says that the last element of the element array which represents a value in the set satisfies $\mathbf{f(x) \geq m}$. This is possible, but complex to implement efficiently.

8.4.1 The 3-fractions Puzzle

As an example of problem-specific symmetry breaking, an in-depth study of the 3-fractions puzzle, a special case of the n-fractions puzzle (CSPLib problem 41), is undertaken. This instance was chosen as there are a number of powerful implied constraints, which have completely perfect GAC implementations under different sets of symmetry breaking constraints. The problem can be formulated in various ways. Definition 8.5 gives both the traditional definition and a slightly different formulation which will be used here.

Definition 8.5. The aim of the 3 fractions puzzle is to find 9 distinct non-zero digits, A to I satisfying:

$$\frac{A}{BC} + \frac{D}{EF} + \frac{G}{HI} \quad = \quad 1 \tag{8.1}$$

Where here and throughout this section, BC is shorthand for $10 * B + C$, *etc.*

An alternative, abstract definition of the 3-fractions puzzle is to find a set variable X of size 3 whose elements are drawn from the following set of tuples:

$$\{\langle a, b, c\rangle | 1 \leq a, b, c \leq 9, a \neq b \neq c \neq a\}$$

and where X satisfies the constraints that no value occurs in more than one tuple in X and $\sum \langle x, y, z\rangle \in X.\ f(x, y, z) = 1$ where $f(x, y, z) = \frac{x}{yz}$.

The second representation given in Definition 8.5 is present to allow the ideas of the previous section to be applied. However to make presentation easier, the discussion of implied constraints will use the first, traditional, model of the problem.

As with most CSPs, there are a number of useful implied constraints which can be found which are not affected by the choice of symmetry breaking. For example, as the digits must be distinct, the following bounds are implied:

$$12 \leq BC \leq 98, \quad 12 \leq EF \leq 98, \quad 12 \leq HI \leq 98 \tag{8.2}$$

$$\frac{A}{BC}, \frac{D}{EF}, \frac{G}{HI} \leq \frac{9}{12} = \frac{3}{4} \tag{8.3}$$

Substituting (8.2) into (8.1), and given that the denominators must be distinct, the following can be derived:

$$A + D + G > 12 \tag{8.4}$$

As the 3 fractions must sum to 1 and are all positive, at least one of the fractions is less than or equal to 1/3, else their sum would be greater than 1. Similarly, at least one of the fractions must be greater than or equal 1/3. These two implied constraints are not practically useful themselves, producing no reduction in the size of search. However, consider breaking symmetry by imposing the following ordering constraints:

$$\frac{A}{BC} \leq \frac{D}{EF} \leq \frac{G}{HI} \tag{8.5}$$

This model, which will be called 'Frac-breaking', is not a complete symmetry breaking method, as for example $1/29 + 3/87 + 4/56$ and $3/87 + 1/29 + 4/56$ are symmetrical, but both satisfy (8.5) since the first two fractions are equal. Nonetheless, (8.5) allows us to make use of symmetry breaking to generate some useful implied constraints.

Using (8.5), $A/BC \leq 1/3$. By similar reasoning, $G/HI \geq 1/3$. Upper and lower bounds can also be derived for D/EF: $1/8 \leq D/EF < 1/2$. The upper bound follows because if $D/EF \geq 1/2$, then $G/HI \geq 1/2$ and $A/BC \leq 0$, which is not possible with non-zero digits. (8.3) implies $A/BC + D/EF \geq 1/4$, as the greatest fraction, G/HI, is $\leq 3/4$. The lower bound follows. Arranging these inequalities into linear form gives:

$$3A \leq BC, \quad 3G \geq HI, \quad 2D < EF, \quad EF \leq 8D \tag{8.6}$$

Simple bounds reasoning on (8.2) and (8.6) now gives, for example, $G \geq 4$ and $H \leq 2$ prior to search, allowing domain reduction in the first node of search. These could not have been generated if either no symmetry breaking was used, or symmetry was broken by simply lexicographically ordering the tuples.

Alternative symmetry-breaking constraints that lead to the derivation of different implied constraints are not discussed. These constraints can potentially reduce the search space even further. For simplicity, consider arranging the variables of the 3-fractions puzzle into a 3×3 matrix: $\begin{smallmatrix} A & D & G \\ B & E & H \\ C & F & I \end{smallmatrix}$. Given the problem constraints, this matrix has column symmetry: any pair of columns can be exchanged in a solution to generate a solution. It does not have row symmetry, so all symmetry can be broken by constraining the columns to be lexicographically ordered [21]. This can be done in six ways, depending on the order of significance of the variables in each column. Three alternatives are considered:

$$\langle A, B, C \rangle \leq_{\text{lex}} \langle D, E, F \rangle \leq_{\text{lex}} \langle G, H, I \rangle \tag{8.7}$$

$$\langle B, C, A \rangle \leq_{\text{lex}} \langle E, F, D \rangle \leq_{\text{lex}} \langle H, I, G \rangle \tag{8.8}$$

$$\langle C, A, B \rangle \leq_{\text{lex}} \langle F, D, E \rangle \leq_{\text{lex}} \langle I, G, H \rangle \tag{8.9}$$

Model LexA uses (8.7). This and AllDifferent({A,B,...,I}) gives:

$$A < D < G \tag{8.10}$$

From (8.4) and AllDifferent({A,B,...,I}), one of $\{A, D, G\}$ is greater than 5. Hence, from (8.10):

$$G > 5 \tag{8.11}$$

Model LexB uses (8.8). This and AllDifferent({A,B,...,I}) gives:

$$B < E < H \tag{8.12}$$

This implies $BC < EF < HI$. Substituting BC for EF and HI in (8.1):

$$A + D + G > BC \tag{8.13}$$

Since the digits are distinct, the left-hand side can be at most $9 + 8 + 7 = 24$. Hence, from (8.13), $B \leq 2$.

Model LexC uses (8.9). This and AllDifferent({A,B,...,I}) gives:

$$C < F < I \tag{8.14}$$

No strong constraints can be derived from (8.14), because C, F and I are the less significant digits in the denominators.

Now the symmetry-breaking models are compared for the 3-fractions puzzle. The 'base' model, from which they are derived, consists of (8.1–8.4) and AllDifferent($A - I$). Any constraint involving rational expressions was multiplied out to avoid rounding errors. This slightly reduces propagation, but all different models had the basic problem constraints implemented identically. Table 8.1 gives a comparison of all the models tested.

For a thorough comparison, the models were compared using all 9! variable orderings of $\langle A, B, C, D, E, F, G, H, I \rangle$. To remove the effects of arriving at a good value ordering by chance, the entire search space is searched for each ordering.

The general version of each implied constraint, for example in the case of (8.11), $\exists \mathbf{X} \in \{\mathbf{A}, \mathbf{D}, \mathbf{G}\}.\ \mathbf{X} > \mathbf{5}$ were tried in all models, but produced uniformly poor performance, as the reduction in search was at most around 30 nodes and constraint solvers typically implement constraints of the types $\exists s \in S.f(s)$ slowly.

	Frac-breaking	LexA	LexB	LexC
Initial Composition	Basic+(8.5)	Basic+(8.7)	Basic+(8.8)	Basic+(8.9)
Best/Worst Choices	910/46,211	895/33,947	253/20,462	849/30,294
Mean Choices	8,041	6,802	2,434	6,859
Implied Constraints	(8.6)	(8.10), (8.11)	(8.12),(8.13)	(8.14)
Best/Worst Choices	668/8,587	812/23,312	198/2,690	734/22,382
Mean Choices	2,507	5,641	734	5,736
Average Improvement	2.39	1.30	2.29	1.19

Table 8.1: Composition of & comparison among models of 3-fractions over all variable orderings.

Table 8.1 reports the number of choices reported by Minion. Individual improvements obtained per model by adding constraints implied by symmetry breaking underlines the importance of choosing the correct method of symmetry breaking based on both the required and known implied constraints in a particular problem. Further, if the models are ranked before and after implied constraints are added, the Fractions-breaking model moves from worst to second best, showing how useful this result can be. This illustrates that often weaker (and in this case incomplete) symmetry breaking can sometimes outperform a stronger scheme when implied constraints are added. Therefore, when choosing a symmetry-breaking scheme, it is important to consider the implied constraints that can be derived. Unfortunately, it is not possible yet to provide hard-and-fast rules as to which particular model should be chosen, because as this problem shows there can be a range of constraints, each of which gains a completely perfect GAC implementation under a different symmetry breaking method.

8.5 Representation Aware Propagation

As Section 8.4 showed, symmetry can cause significant problems during the solving of CSPs. This is caused by limiting propagation as well as the previously identified problem of causing the multiple symmetrically equivalent search trees to be explored. The previous section showed how symmetry breaking constraints can be tailored to particular problem constraints, making previously poorly propagating constraints gain a completely perfect GAC implementation in a representation with symmetry breaking.

This system does however have its limitations. It suffers from the usual problems with static symmetry breaking constraints, such as interacting poorly with certain variable heuristics. Further, it requires choosing in advance a particular constraint or constraints to give a completely perfect GAC implementation, and can not be dynamically altered during search.

A more general method would be to weaken the requirements on propagators, so they only had to ensure at least one representative of each assignment to the original domain was not removed, rather than the current requirement that every assignment which represents a possible solution is not removed. This would allow the kind of deduction given in Examples 8.4 and Example 4.8 on page 62, which appears both correct, and to subsume the previous section.

Example 8.4. *Consider a constraint problem which consists of finding a set S of size 2 and a set T of size 4 drawn from $\{1, 2, 3, 4, 5\}$, which are represented by the explicit representation without symmetry breaking by element arrays $\overline{S'}$ and $\overline{T'}$. The only constraint in the CSP is $S \subseteq T$, which is equivalent to the constraint $\forall i \in \{1, 2\}.\ \exists j \in \{1, 2, 3, 4\}.\ S'[i] = T'[j]$ on the representations of S' and T'.*

Assume the first branch made in search is $S'[1] = 3$. At this point it is clear that some element of $\overline{T'}$ must be assigned 3 in all solutions. However as it could be any of the variables in $\overline{T'}$ which is assigned 3, no pruning can occur.

Since it is known at least one variable must be assigned the value 3, by symmetry some variable could be chosen and assigned 3. This would clearly remove some solutions, but it would leave at least one symmetric image of each solution.

There are a number of basic problems with changing propagation in this way which must be overcome. Firstly, as Example 4.8 shows, such propagators are not monotonic. Therefore the order in which they are applied has an effect on the resulting search state. Further, Example 5.2 shows how for only a single constraint there may be distinct, incomparable minimal sub-domains which represent a required list of assignments. Despite these difficulties however, applying any representation aware propagation must result in a smaller search space than not doing so, and therefore it is still worthwhile finding cases where it can be applied.

As it is not possible to implement it over all constraints at the same time, representation aware propagation is not a replacement for other forms of symmetry breaking, but instead complements it. It cannot in general be combined with static symmetry breaking, as this would create a situation identical to the previous section, where the choices of which assignments to keep was made before search began. However, it does work well with dynamic symmetry breaking methods.

Consider Example 8.4. Rather than simply assigning the first variable in $\overline{T'}$ the value 3, a branch could make this variable 3 on one branch, and not 3 on the other. This would have created an identical effect down one branch of the search tree, but produced a new, search tree which explored the first variable of $\overline{T'}$ not being 3, where there would be no solution. Further, a propagating dynamic symmetry breaking method will not search this subtree at all, as shown in Example 8.5.

Example 8.5. *Consider a CSP which contains an array $\overline{T}$ of 4 symmetric variables each of domain $\{1, 2, 3\}$, where this array has complete variable symmetry.*

Assume the branch $T[1] = 1, T[1] \neq 1$ is performed, and the whole of the left hand branch is searched, and then the right branch is begun.

This leaves the domains $T[1] \in \{2, 3, 4\}, T[2], T[3], T[4] \in \{1, 2, 3, 4\}$. Consider the assignment $T[2] = 1$. Then under the symmetry which swaps $T[1]$ and $T[2]$, this gives $T[1] = 1, T[2] \in \{2, 3, 4\}, T[3], T[4] \in \{1, 2, 3, 4\}$. This is subsumed by the search node first reached after $T[1] = 1$ was performed, as there $T[2]$ also had 1 in its domain.

Therefore a dynamic symmetry breaking algorithm can legally delete the assignment $T[2] = 1$, and similarly the assignments $T[3] = 1$ and $T[4] = 1$, leaving $T[1], T[2], T[3], T[4] = \{2, 3, 4\}$.

Example 8.5 shows at least some representation aware propagation can be performed without the need to alter how solvers propagate or deal with symmetry with only a small increase in search size, by introducing some new nodes which instantly fail. This is accomplished by a dynamic symmetry breaking method and careful choice of variable branching. This may at first seem as if it implies representation aware propagation is not useful, however this idea of search heuristics and propagation methods being interchangeable is not new. For example, Singleton Arc Consistency (SAC) [17] is a propagation method which removes any domain value which if assigned would lead to GAC propagation failing. However, this can be completely implemented just using GAC with only a linear increase in search size, by looking at all literlas which SAC would deleted, and then branching on each of these literals which will immediately backtrack. This does not stop SAC being a powerful and useful propagation method, as the only known way of finding such values is to perform the various SAC algorithms.

Implementing representation aware propagators for arbitrary constraints and representations is beyond the scope of this thesis. As a proof of principle a common special case will be considered. Consider the n-way limited *explicit* representation where the only allowed propagation is to assign an unassigned variable. At a high level, this involves finding when a particular value must occur in the set being represented but is not currently assigned to any variable, and then assigning a variable that value. Example 8.6 gives a practical example of this idea, and how it can be used to massively reduce the size of search performed.

Example 8.6. *Consider the constraint problem with three variables A,B and C, where A and B have the domain $\{s \mid s \subseteq_m \{1, \ldots, n\}, |s| = p\}$ and C has the domain $\{s \mid s \subseteq_m \{1, \ldots, n\}, |s| = p\}$, where $1 < p < n$, and the single constraint $A + B = C$.*

A, B and C are represented by the explicit representation, where A,B and C have element arrays A', B' and C' respectively. Note that there is no need for a check array as the sets are of fixed size, and all assignments to the element array represent valid multisets.

If these arrays were searched by going first through A', then B' and finally C' and dynamic symmetry breaking was used, it is possible to see that the only propagation which can occur is removing values from C' which are not assigned to any element of A' or B'. Until however C' is itself assigned there is no way the search can fail as there is no way any domain can become empty. Therefore the search space will be at least of size 2^{2p}

When symmetry breaking propagation is used however, whenever a variable in A' is assigned it is clear that a variable in C' can be assigned the same value. Therefore the search will fail when the last element of A is assigned, as this value will not be assignable to C'. The search space will therefore be at most size 2^p.

In fact, using the n-way limited *explicit* representation with both representation aware propagation and dynamic symmetry breaking is identical to the *occurrence* representation. Both can represent a value being forbidden from the set and both also represent a value being propagated to always be in the set. Furthermore, neither allows any other form of propagation. Dynamic symmetry breaking ensures multiple symmetric images of the same partial assignment are not checked.

8.6 Conclusion

Correctly handling symmetry is an important part of constraint programming. By storing symmetries in a structured, nested manner rather than considering the symmetry of the whole CSP as a single group it is often possible to use the structure to handle different levels of symmetry seperately, leading to both easier and more efficient implementations. Further, by careful analysis of symmetric representations and the constraints on them it is possible to design specialised symmetry breaking constraints which can improve propagation. This provides large improvements in performance.

Finally, by understanding how symmetry of representations can cause a loss of propagation this chapter has explained and shown a proof of principle of a new symmetry breaking method, which proactively rather than reactively breaks symmetry. This can lead to exponential improvements in search size over traditional symmetry breaking methods.

Chapter 9

Conclusion

This chapter summarises the strengths and limitations of this thesis and also outlines future work which the author hopes will usefully extend the results of this thesis. Section 9.1 provides a summary of novel contributions this thesis has made. Section 9.2 describes the limitations and flaws in this thesis and how they might be overcome in future work.

9.1 Summary

The aim of this thesis is to provide a useful method of comparing very different representations of high-level types while avoiding implementation details wherever possible. This thesis has largely achieved that goal. The definition of representations given in Chapter 4 encapsulates both replacing a single high-level variable with a series of variables and the existing special data structures solvers use to store these domains. The existing solving frameworks of propagators and branching have been mapped onto the definition of representations to give a framework in which they can be studied.

This work has a number of important applications, in particular it provides a solid framework on which to build an automated refinement system. Such a system could apply the rules provided by this thesis automatically. This could solve many of the huge number of choices such a system would have to make, if allowed to refine and implement a problem using all of the techniques currently in use in CP. While many such choices would be obvious to an expert practitioner in the field, they must still be formalised and presented in a form suitable for a computer.

Chapter 5 defines one of the strongest possible relationships two representations can have, that one always outperforms the other in terms of search space. In the case of variables, a number of common operations are shown to generate dominating variable representations and this is used to show relationships between a number of representations of sets. This chapter also shows that many common alternative representations are equivalent, despite the fact that previous research has shown differences in their performance. This is shown to arise mainly from limitations in current solver implementations, rather than a theoretical limitation. However, many representations are still incomparable under this framework, despite the fact it may be possible to compare them on particular problems.

Chapter 6 considers comparing representations on languages of constraints rather than over all CSPs. This chapter demonstrates the most practically useful result of the thesis, that the *occurrence* representation performs as well as the complete representation on a large range of set and multiset constraints, including subset, union, intersect and element. This provides a new method of discussing how well representations perform which is stronger than simply that GAC is achieved on the constraints on the representation, but instead compares the representation against the performance of all possible other repersentations. The only common constraints excluded are those involving the cardinality of (multi)sets and disequality constraints.

The most commonly used constraint on which the *occurrence* representation does not perform as well as the complete representation is the cardinality constraint. It is shown that a small set of constraints which includes cardinality and a selection of the common set constraints can not have an efficient implementation. This is because any representation with completely perfect GAC implementation of all the constraints requires the implementation of the propagators of at least one of these constraints to be NP-hard. This provides a strong theoretical limit on how powerful such representations can be in practice if they are to remain efficient.

Chapter 7 differs from the other chapters in the thesis by providing a number of experiments which show some important features of the theory of representations. These experiments are not meant to provide comprehensive testing of all representations in this thesis, rather to illustrate a few important concepts. There are three important conclusions which can be drawn from these experiments:

- The *occurrence* representation outperforms a randomly generated model on problems involving many set and multisets constraints on which it is has a completely perfect GAC implementation, even when the problem involves other constraints.

- The *occurrence* and random representations perform identically well on problems whose constraints are generated randomly.
- In general, channelling is very hard to implement in such a way that it can improve performance of a selection of models. Given a selection of models and no method of knowing which is best, it is almost impossible to construct a model that can compete with a single good model in isolation.

The first two points back up the earlier chapters of the thesis, showing that when comparing representations where no dominance hold, they must be compared with respect to the constraints in the CSP being modelled. The experiments on channelling show that it is not possible to escape categorising and comparing models by simply attempting to use many models at once and channel them together, instead the models must be categorised as to their usefulness in isolation.

Chapter 8 explores the problems of symmetry in representations. It demonstrates that the symmetries of representations form a useful subclass of symmetries which can be handled efficiently. In particular, when representations are applied in a nested fashion, the resulting symmetry can also be handled in a nested fashion, improving performance and simplifying implementation.

This chapter also shows that symmetry in representations can often be broken in different ways, some of which will line up much better with the problem constraints than others. Further, this method can be generalised to a dynamic method of proactively breaking symmetry, rather than breaking it reactively, which can improve performance and in particular produce a case where the two main set representations, the *explicit* and *occurrence*, become equivalent.

9.2 Limitations and Future Work

While this thesis has presented a number of results there are a number of areas where this work can be extended and improved. Many of these relate to the fact that this thesis has avoided any results which cannot be backed with theory and require experiments or heuristic measures. This is important, as this means that assuming they are correctly applied, any results of this thesis could be applied by an automated modelling system without the need to worry that a particular heuristic may not apply to a particular problem. This has however still left a number of areas for future work. These fall into four main areas.

Firstly, while the basic definitions of representations and propagators can apply to any type of high-level variable, for space and time reasons they have only been applied to sets and multisets. An obvious immediate requirement is to try applying the same theory and principles to representations of other high-level types, such as functions, relations and graphs. Some preliminary work suggests in many cases this is fairly simple, in particular because many representations for these high-level types have a strong resemblance to the *occurrence* and *explicit* representations of sets and multisets. Another, more complex route would be to apply the theory to less mathematical and more practical high-level types, such as a schedule or bin packing problem. How productive this would be is unclear.

Secondly, the examples presented and some parts of the theory have ignored propagation other than GAC. There are a number of ways in which the work could be extended to consider other levels of propagation, some of which would dovetail with existing work. Obvious examples of this would involve seeing where propagators can be implemented in polynomial time and where simpler forms of propagation are equivalent to GAC. Further, considering if a representations has a completely perfect implementation with respect to some form of propagation other than GAC would be more complex, but possible.

Thirdly, channelling and joining representations has been looked at in only a preliminary fashion. The two main results in this area, showing how joining equivalent representations is not useful and channelling between very different representations in general leads to worse performance, are both important. However, it may still be that the non-equivalent representations which arise in practice can still be usefully channelled.

Finally, possibly the most major limitation of this thesis is a lack of immediate practical application to choosing representation in real-world problems containing many constraints. While it is possible to find problems where some representation is perfect, there has been no attempt to discuss what should be done in other cases. Attempting to encapsulate a “proportion of perfectness” for a representation would allow the theory to be applied more widely. There are a number of problems with trying to construct such a theory.

Preliminary experiments suggest trying to measure this with a single numerical value, for example the proportion of values left after propagation, works well for random representations and random constraints, but performs very badly on more structure representations and constraints. Further, constraints which are not symmetric, so as lexicographic ordering, can perform very differently depending on what order variables are assigned. This must be researched however, as heuristics in this area would appear a necessity to be able to implement an automated refinement system.

List of References

[1] *KIDS: A Knowledge-Based Software Development System*, 1991.

[2] L. Abraido-Fandino. An overview of REFINE^{TM} 2.0. In *Proceedings of the Second International Symposium on Knowledge Engineering, Madrid, Spain*, 1987.

[3] K. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[4] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In *Proceedings of CP '99*, pages 73–87, 1999.

[5] N. Barnier and P. Brisset. Solving the kirkman's schoolgirl problem in a few seconds. In *Proceedings of CP '02*, 2002.

[6] Christian Bessière, Emmanuel Hebrard, Brahim Hnich, and Toby Walsh. Disjoint, partition and intersection constraints for set and multiset variables. In *Proceedings of CP '04*, 2004.

[7] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artificial Intelligence*, 165(2):165–185, 2005.

[8] James R. Bitner and Edward M. Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

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

[10] Andrei Bulatov, Peter Jeavons, and Andrei Krokhin. Classifying the complexity of constraints using finite algebras. *SIAM Journal of Computing*, 34(3):720–742, 2005.

[11] M. Cadoli, G. Ianni, L. Palopoli, A. Schaerf, and D. Vasile. NP-SPEC: An executable specification language for solving all problems in NP. *Computer Languages*, 26:165–195, 2000.

[12] A.M. Cheadle, W. Harvery, A. J. Sadler, J. Schimpf, K. Shen, and M.G. Wallace. ECLiPSe: An introduction. Technical Report IC-Parc-03-1, Imperial College London, 2003.

[13] B. M. W. Cheng, Kenneth M. F. Choi, Jimmy Ho-Man Lee, and J. C. K. Wu. Increasing constraint propagation by redundant modeling: an experience report. *Constraints*, 4(2):167–192, 1999.

[14] Stephen Cook and Robert Reckhow. On the lengths of proofs in the propositional calculus (preliminary version). In *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 135–148, New York, NY, USA, 1974. ACM Press.

[15] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM Press.

[16] J. Crawford. A theoretical analysis of reasoning by symmetry in first-order logic. In *AAAI Workshop on Tractable Reasoning*, 1992.

[17] Romuald Debruyne and Christian Bessière. Some practicable filtering techniques for the constraint satisfaction problem. In *Proceedings of IJCAI '97*, pages 412–417, Nagoya, Japan, 1997.

[18] Grégoire Dooms, Yves Deville, and Pierre E. Dupont. CP(Graph): Introducing a graph computation domain in constraint programming. In *Proceedings of CP '05*, pages 211–225, 2005.

[19] Iván Dotú, Alvaro del Val, and Manuel Cebrián. Redundant modeling for the quasigroup completion problem. In *Proceedings of CP '03*, pages 288–302, 2003.

[20] Niklas Een and Niklas Sörensson. An extensible SAT-solver [ver 1.2].

[21] P. Flener, A.M. Frisch, B. Hnich, Z. Kiziltan, IM, J. Pearson, and T. Walsh. Breaking row and column symmetries in matrix models. In P. van Hentenryck, editor, *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming*, pages 462–476, 2002.

[22] P. Flener, J. Pearson, and M. Agren. Introducing ESRA, a relational language for modelling combinatorial problems. In *Proceedings of LOPSTR '03: Revised Selected Papers*, volume 3018 of *LNCS*.

[23] Pierre Flener, Alan M. Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Exploiting common patterns in constraint programming. In *International Workshop on Reformulating Constraint Satisfaction Problems*, 2002.

[24] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

[25] Eugene C. Freuder. Synthesizing constraint expressions. *Communications of the ACM*, 21(11):958–966, 1978.

[26] Alan Frisch, Brahim Hnich, Zeynep Kiziltan, Ian Miguel, and Toby Walsh. Global constraints for lexicographic orderings. In *Proceedings of CP'02*, 2002.

[27] Alan M. Frisch, Matthew Grum, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The design of essence: A constraint language for specifying combinatorial problems. In Manuela M. Veloso, editor, *Proceedings of IJCAI '07*, pages 80–87, 2007.

[28] Alan M. Frisch, Christopher Jefferson, Bernadette Martínez Hernández, and Ian Miguel. The rules of constraint modelling. In *Proceedings of IJCAI '05*, 2005.

[29] A.M. Frisch, B. Hnich, Z. Kiziltan, I. Miguel, and T. Walsh. Multiset ordering constraints. In *18th International Conference in AI*, 2003.

[30] A.M. Frisch, C. Jefferson, and I. Miguel. Constraints for breaking more row and column symmetries. In *Proceedings of CP '03*, 2003.

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

[32] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1990.

[33] Pieter Andreas Geelen. Dual viewpoint heuristics for binary constraint satisfaction problems. In *ECAI '92: Proceedings of the 10th European conference*

on Artificial intelligence, pages 31–35, New York, NY, USA, 1992. John Wiley & Sons, Inc.

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

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

[36] Ian P. Gent, Christopher Jefferson, and Ian Miguel. Minion: A fast scalable constraint solver. In *Proceedings of ECAI '06*, pages 98–102, 2006.

[37] Ian P. Gent, Tom Kelsey, Steve Linton, Iain McDonald, Ian Miguel, and Barbara M. Smith. Conditional symmetry breaking. In Peter van Beek, editor, *CP*, volume 3709 of *Lecture Notes in Computer Science*, pages 256–270. Springer, 2005.

[38] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In Werner Horn, editor, *Proceedings of ECAI 2000*, pages 599–603. IOS Press, 2000.

[39] Carmen Gervet. Conjunto: constraint logic programming with finite set domains. In Maurice Bruynooghe, editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 339–358, Massachusetts Institute of Technology, 1994. The MIT Press.

[40] W. Harvey and P.J. Stuckey. Improving linear constraint propagation by changing constraint representation. *Constraints*, 8[2]:173–207, 2003.

[41] Warwick Harvey. Symmetry breaking and the social golfer problem. In *Proceedings of SymCon'01: Symmetry in Constraints*, pages 9–16, 2001.

[42] P. Van Hentenryck. *The OPL optimization programming language.* The MIT Press, 1999.

[43] Pascal Van Hentenryck, Yves Deville, and Choh-Man Teng. A generic arc-consistency algorithm and its specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992.

[44] B. Hnich, B.M. Smith, and T. Walsh. Dual modelling of permutation and injection problems. *Journal of Artificial Intelligence Research, Volume 21.*

[45] Brahim Hnich. *Function Variables for Constraint Programming.* PhD thesis, Uppsala, 2004.

[46] Joey Hwang and David G. Mitchell. 2-way vs. d-way branching for CSP. In *Proceedings of CP '05*, pages 343–357, 2005.

[47] ILOG S.A. *ILOG Solver 5.3 Reference and User Manual*, 2002.

[48] Chris Jefferson, Tom Kelsey, Steve Linton, and Karen Petrie. Gaplex: Generalized static symmetry breaking. In Frédéric Benhamou, Narendra Jussien, and Barry O'Sullivan, editors, *Trends in Constraint Programming*, chapter 9, pages 191–205. ISTE, London, UK, May 2007.

[49] Zeynep Kiziltan and Toby Walsh. Constraint programming with multisets. *Proceedings of the 2nd International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon-02)*, 2002.

[50] Vitaly Lagoon and Peter J. Stuckey. Set domain propagation using ROBDDs. In *Proceedings of CP '04*, 2004.

[51] Y. C. Law and J. H. M. Lee. Algebraic properties of CSP model operators. In *Proceedings of the International Workshop on Reformulating Constraint Satisfaction Problems: Toward Systematisation and Automation*, pages 57–71, 2002.

[52] Y. C. Law and J. H. M. Lee. Model induction : a new source of CSP model redundancy. In *AAAI*, 2002.

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

[54] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, pages 245–250, 2003.

[55] E. Luks and A. Roy. Symmetry breaking in constraint satisfaction. In *Intl. Conf. of Artificial Intelligence and Mathematics*, 2002.

[56] Alan K. Mackworth. Consistency in networks of relations. Technical report, Vancouver, BC, Canada, Canada, 1975.

[57] C. Mears, M. Garcia de la Banda, and M. Wallace. On implementings symmetry detection. In *Proceedings of SymCon'06*, 2006.

[58] Michela Milano and Willem J. van Hoeve. Reduced cost-based ranking for generating promising subproblems. In *Proceedings of CP '02*, pages 1–16, London, UK, 2002. Springer-Verlag.

[59] Roger Mohr and T.C. Henderson. Arc and path consistency revisited. *Artificial Intelligence*, 28:225–233, 1986.

[60] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Inf. Sci.*, 7:95–132, 1974.

[61] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[62] Karen E. Petrie. *Constraint Programming, Search and Symmetry.* PhD thesis, University of Huddersfield, 2005.

[63] Steve Prestwich and Andrea Roli. Symmetry breaking and local search spaces. In *Second International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2005.

[64] J.-F. Puget. On the satisfiability of symmetrical constraint satisfaction problems. In *Methodologies for Intelligent Systems (Proceedings of ISMIS'93)*, volume 689 of *LNAI*, pages 350–361. Springer, 1993.

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

[66] Colin R. Reeves, editor. *Modern heuristic techniques for combinatorial problems.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[67] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of 12th National Conference on AI (AAAI'94)*, volume 1, pages 362–367, 1994.

[68] Andrea Rendl, Ian P. Gent, and Ian Miguel. Tailoring solver-independent constraint models: A case study with essence' and minion. In *Proceedings of SARA 07*, 2007.

[69] Francesca Rossi, Charles Petrie, and Vasant Dhar. On the equivalence of constraint satisfaction problems. In Luigia Carlucci Aiello, editor, *ECAI '90*, pages 550–556, Stockholm, 1990. Pitman.

[70] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., New York, NY, USA, 2006.

[71] Andrew Sadler and Carmen Gervet. Hybrid set domains to strengthen constraint propagation and reduce symmetries. In *Proceedings of CP '04*, pages 604–618, 2004.

[72] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming With Sets : An Introduction to SETL*. Springer-Verlag, 1986.

[73] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. *Discrete Applied Mathematics*, 2002.

[74] B. Smith. Reducing symmetry in a combinatorial design problem. Technical report, University of Leeds, 2001. RR 01, University of Leeds (UK), School of Computer Studies, 2001.

[75] The Geocode team. Generic constraint development environment.

[76] Toby Walsh. Consistency and propagation with multiset constraints: A formal viewpoint. In *Proceedings of CP '03*, 2003.

[77] Stephen J. Westfold and Douglas R. Smith. Synthesis of efficient constraint satisfaction programs. *Knowledge Engineering Reviews*, 2001.