The University of York Department of Computer Science

Submitted in part fulfilment for the degree of MMath.

Attacking the Grid Colouring Problem with Constraint Programming

Christopher Bak

February 8, 2012

Supervisor: Dr. Alan Frisch

Words: 15,823, as counted by detex Report.tex | wc -w
This includes the body of the report but not the bibliography.

Abstract

This project aims to solve the 17×17 4-colouring challenge first proposed by William Gasarch in late 2009. The challenge is to fill a 17×17 grid with four colours such that no rectangle within the grid has the same colour in all of its four corners. This is one of the few unsolved problems in grid-colouring, and it has been studied from several different perspectives since the challenge was made. Even if the problem remains unsolved, this project should a good insight into the grid-colouring problem from a constraint programming point of view.

Constraint programming will be used to attack this problem, a method that has not been previously attempted in any great detail. A Constraint Satisfaction Model is built for the general 4-colouring problem, and a ground-up approach is taken; as the model is improved, it is able to find larger grid-colourings in a reasonable amount of time. Both common constraint programming techniques and properties of the problem are used to enhance the model. This project makes use of MiniZinc, a modern and succinct constraint modelling language which is able to capture the problem in a concise manner.

Contents

1 Introduction

On November 30, 2009 William Gasarch posted a blog [1] in which he offered \$289 to the first person to find a 4-colouring of a 17×17 grid, under the assumption that finding such a 4-colouring is possible. A k-colouring of a particular grid is defined to be a labelling of each cell of the grid with one of k colours such that for each rectangle R in the grid, all four corners of R are not the same colour. If a grid has a k-colouring, then it is said to be k-colourable.

Fenner et al. [2] have thoroughly studied k-colourings of grids up to 4-colourings using a variety of mathematical concepts. In particular, they proved that all but four grids, up to rotation, are either 4-colourable or not. They are 12×21, 17×17, 18×17 and 18×18. Gasarch posed the challenge for 17×17 in particular because he strongly believes that it is 4-colourable [1]. Outside of the aforementioned paper there is little published material on the problem, although a near solution has been constructed by Rohan Puttagunta, with only one grid element preventing a full solution [1]. Indeed, a logical construction of a solution is a good approach to solving this problem. There are $4^{289} \approx 10^{174}$ ways to fill 17×17 grid with four colours, so this problem is clearly infeasible to solve using brute force.

This project will make use of constraint programming to attempt to find a solution. Grid-colouring is a non-arithmetic problem that has a natural matrix model, like other famous combinatorial problems such as Sudoku or N-queens. The constraints for these problems restrict the arrangement of the values, as opposed to problems with arithmetic constraints, such as the magic square problem. Constraint programming is a very nice way of solving these problems, so it should prove effective at solving the general grid-colouring problem. To the best of my knowledge, no serious constraint programming attempt has been made on this problem. This project uses MiniZinc, a modern constraint modelling language. Constraint solvers do rely on search, and the search space for this problem has exponential growth, but by appropriate modelling of the problem and through applying constraint programming techniques, the search space will be reduced considerably. If 17×17 is not 4-colourable, then any search-based approach will almost certainly not prove that the problem is unsolvable, as the entire search space will have to be explored to achieve this. On the other hand, if it is 4-colourable, only one solution is required, and finding one solution will take considerably less work.

The background to the problem and the necessary preliminaries are in Chapter 2. The main constraint of the problem is analysed in Chapter 3. The symmetries of the problem are identified and dealt with in Chapter 4. Extensions and an alternative model are considered in Chapters 5 and 6. The evaluation of the project and discussion of further work are in Chapters 7 and 8.

1.1 Statement of Ethics

This project is empirical, and neither the methods undertaken nor the results of this project have an impact on the lives of any people, so there are no ethical issues to be considered.

2 Background

Throughout the rest of this paper, the notation $N \times M$ will be used to denote the $N \times M$ 4-colouring problem.

2.1 The Constraint Satisfaction Problem

The Constraint Satisfaction Problem (CSP) is a common way of representing problems, defined using a set of variables that must satisfy a number of constraints. Formally, a CSP is a triple (X, D, C) where:

> X is a finite set of variables $\{x_1, \ldots, x_n\}$.
>
> D is a finite set of domains $\{D_1, \ldots, D_n\}$ where D_i corresponds to the set of values to which the variable x_i can be assigned. For the purposes of this project, it is assumed that $|D_i| < \infty \; \forall 1 \leq i \leq n$. Such problems are known as finite domain CSPs.
>
> C is a finite set of constraints. Each $c \in C$ is a pair (S, R) where $S \subseteq X$ is a finite set of variables $\{x_1, \ldots, x_k\}$ called the scope of c, and R is a relation on these variables, or a subset of $D_1 \times \ldots \times D_k$.

A constraint c is satisfiable if for every $x_k \in S$ there are values $v_k \in D_k$ such that $(v_1, \ldots, v_k) \in R$. Then the assignment $x_i = v_i \; \forall \; 1 \leq i \leq k$ satisfies c. A total assignment $x_i = v_i \; \forall \; 1 \leq i \leq n$ is a solution of the CSP if it satisfies every constraint in C.

To illustrate this, consider 17×17. X is be the set of 289 variables representing each cell in the grid. The domains of each variable are equivalent, a set of four colours, say $D = \{red, blue, green, yellow\}$. For each set of four variables Rec representing a rectangle, there is a constraint c with Rec = S, where S is the set of all 4-tuples containing elements of D except for the four 4-tuples whose elements are all equal.

2.2 Generalised Arc Consistency

Let c be a constraint with $|S| = k$. c is generalised arc consistent (GAC) if for every $x_i \in S$ and $v_i \in D_i$, there exist values $v_j \in D_j \; \forall \; 1 \leq j \neq i \leq k$, such that the assignment $x_j = v_j$ satisfies c.

If a constraint is not GAC, then there exists a value $v_i \in D_i$ for some i which can never satisfy c and will therefore never participate in any solution of the CSP. Hence v_i can be removed from D_i without consequence. Removing all such values from the domains of their

corresponding variables will ensure the constraint is GAC. This process is called propagation. Efficient propagation is very important, and a lot of constraint programming research deals with developing algorithms for common constraints to maximise propagation speed [3].

2.3 Search

The CSP is, in essence, a search problem. Assuming all domains have equal size d, for a CSP with n variables, there are n^d possible total assignments, and the set of solutions is a (possibly empty) subset of these. A typical CSP will have many solutions. Depending on the problem specification, it might be necessary to find all solutions, or to find a "good enough" solution, usually by minimising or maximising a cost function. For this problem, however, it is sufficient to find only one solution or to prove unsatisfiability.

Two important factors for a constraint programmer to consider are variable ordering and value assignment. In what order should variables be assigned values, and how will those values be selected from the domain to maximise search efficiency? The answers to these questions are problem-specific, but it is often a sensible choice to first assign values to the variables which are constrained the most so that propagation can occur early in the search, where backtracking isn't as costly.

The solver combines search with propagation. This can be done in several ways, which could differ depending on the solver, but the general idea is the same. The search starts with all variables uninstantiated. Variables are assigned values according to the chosen variable ordering and value assignment. If a variable assignment causes all the variables in the scope of any constraint to be instantiated, then that partial assignment is checked to see if it satisfies the constraint. If not, then the most recent assignment is undone and the associated value is removed from the domain of that variable. This is called backtracking. Look-ahead can also be used during search to various degrees. For example, when a variable x is assigned a value, propagation could be performed on all unassigned variables. This results in possible domain reduction for an unassigned variable y depending on the value assigned to x, where x and y are in the scope of the same constraint.

2.4 Introduction to MiniZinc

MiniZinc is the constraint modelling language that will be used in this project. It was created in Melbourne in order to unify the large number of different constraint solvers that exist, which had their own, incompatible and often vastly different, constraint modelling languages [4] [5]. This is still a work in progress, but MiniZinc is currently supported by a few constraint solvers, including Eclipse and Gecode [4]. Some basic concepts of MiniZinc are presented in this section to familiarise the reader with the language. The MiniZinc model in Figure 2.1 is presented as an illustrative example. It will produce an 3×3 grid containing distinct integers from the set $\{1, \ldots, s^2\}$ such that each row is in strictly increasing order.

```
1  include "globals.mzn";
2  par int: s = 3;
3  % int: s = 3;
4  set of int: Size = 1..s;
5  array[Size,Size] of var 1..pow(s,2): grid;
6
7  constraint forall(i in Size, j in 1..s-1) (
8   grid[i,j] < grid[i,j+1]
9  );
10 constraint
11 alldifferent([grid[i,j] | i in Size, j in Size]);
12
13 solve satisfy;
14
15 output [show(grid[i,j]) ++ if j mod s = 0
16 then "\n" else " " endif | i in Size, j in Size]
```

Figure 2.1: A simple MiniZinc model to fill an $s \times s$ grid.

2.4.1 Variables

MiniZinc has two kinds of variables: parameters and decision variables. Parameters are variables that must have a fixed value assigned to them when the model is compiled. This is done by using the equivalence relational operator = . In this example, the single parameter s defines a problem instance. A different problem instance is represented by a different sized square grid. If the user wishes to find a 4×4 grid, the user only needs to change the assignment of s to 4.

Decision variables are the variables of the CSP. They can be assigned values in the same way as parameters, but they are not required to be fixed at compile time. Instead, the variables are assigned values when the solver searches for a solution under the restrictions of the domains and constraints specified in the MiniZinc model. The values of decision variables can change during search, due to backtracking.

The keywords **par** and **var** are used to declare parameters and decision variables respectively. MiniZinc will create a parameter by default if the keyword is omitted, so the use of **par** is not strictly necessary. This is illustrated in lines 2 and 3 of Figure 2.1. Both declarations are equivalent. Note that the % sign is used to comment out line 3.

MiniZinc supports a small number of basic types, namely integers, floats, strings and booleans. Compound types can be constructed from basic types. For example, sets of basic types can be declared. A set is a collection of distinct values of a particular type. Sets can be assigned values in several different ways. The set named Size is assigned values using a range expression of integers in line 4. Alternatively, set literals can be used, i.e. $\{1, 2, 3, 4\}$, or even set comprehensions.

MiniZinc also supports the familiar array structure. Arrays can contain either parameters,

variables or sets, but they cannot be indexed by variables. Unlike many other programming languages, the indices of an array must be specified as an integer range expression as opposed to a single integer value. A two-dimensional array named $grid$ containing decision variables in the domain $\{1, \ldots, s^2\}$ is declared in line 5. The arithmetic operator *pow* is used to calculate s^2. Array elements are accessed by typing the name of the array followed by the indices in square brackets. For example, $grid[1, 1]$ is the element in the first row and first column of the array grid. Array comprehensions can also be used in the same way as set comprehensions, the only difference being that set comprehensions have curly braces while array comprehensions have square braces.

2.4.2 Constraints

A constraint in MiniZinc is a boolean expression preceded by the keyword **constraint**. The solver will assign values to variables in order to satisfy all the constraints. Satisfying a constraint is equivalent to making its boolean expression true. Assigning a value to a decision variable is equivalent to constraining that variable to have that value, thus the following two code fragments are equivalent:

```
Fragment 1: var int: x = 3;

Fragment 2: var int: x;
            ...
            constraint(x = 3);
```

A global constraint is a constraint that acts on an unspecified number of variables. MiniZinc has a number of pre-defined global constraints. An example is *alldifferent*, a very common constraint in CSPs. As the name suggests, this constraint is satisfied if no two variables in its scope take the same value. This constraint is used in line 11. An array comprehension is used to flatten the two-dimensional array *grid* into a one-dimensional array since *alldifferent* must take a one-dimensional array as its argument. All global constraints are defined in the file globals.mzn, which can be included in the model with the keyword **include**. Global constraints are a powerful feature of MiniZinc. Firstly, they provide a quick and elegant way of expressing frequently used global constraints that could otherwise be difficult or inelegant to write. Secondly, most of these constraints are well studied and will often have their own powerful propagation algorithms used by the constraint solvers. It is also possible for the user to define their own constraints using the keyword **predicate**. An aggregation operator *forall* is used in the second constraint to traverse through each row of the array. There are other aggregation operators defined both for arrays and for sets.

2.4.3 Solving and output

The keyword **solve** is used to tell MiniZinc what to search for and how to search for it. The example model contains a basic solve item, *solve satisfy*, which instructs MiniZinc to return

the first solution it finds using the default search strategy. MiniZinc can also search for a solution which maximises or minimises an arithmetic expression using *solve maximise* {*expr*}.

Search annotations are used to specify a search strategy. A search annotation is attached to the solve item by the operator ::, and it has the following format.

```
solve :: search_type(array, varchoice,
         constrainchoice, search_strategy) satisfy;
```

The search type specifies the type of decision variable to search for. There are currently three search types: *int_search*, *bool_search* and *set_search*; which correspond to the types integer, boolean, and set of integers respectively. The first argument is a one-dimensional array of decision variables whose type matches the search type. The second argument specifies the variable order. The third argument determines the choice of value to assign each variable. The final argument gives the search strategy, which will always be *complete*.

An example of a search annotation is

```
solve :: int_search([grid[i,j] | i in Size, j in Size], input_order,
         indomain_min, complete) satisfy;
```

Once again, $grid$ has been flattened since two-dimensional arrays are not allowed in a search annotation. The order of the variables in the comprehension has an impact on the way the array is flattened. Both i and j have the same range, namely $1, \ldots, s$. Since i appears first in the comprehension, the solver will fix $i = 1$ before working its way through all possible values of j. Once these values have been exhausted, $i = 2$ will be fixed and the process is repeated. Hence the one-dimensional array is constructed by taking all the values in the first row from left to right, appending the second row to it, and so on. In contrast, the array comprehension

```
[grid[i,j] | j in Size, i in Size]
```

evaluates to a one-dimensional array constructed by appending the columns of $grid$ together from left to right. The second argument is *input_order*, which means the variables will be chosen in order from the array in the first argument. *indomain_min* will instruct the solver to assign to the current decision variable the smallest value from its domain.

The final component of a MiniZinc model is the output item. If a solution has been found, all decision variables will have been instantiated. The keyword **output** followed by a list, or one-dimensional array, of string expressions tells MiniZinc how to display the desired output on the standard output. C-like character literals are used to print white space characters. Conditional expressions and array comprehensions are also allowed in the output statement. The operator *show* converts decision variables or parameters into strings and the operator ++ is used to concatenate multiple strings.

2.5 Previous Work

The general grid colouring problem was studied in great detail by Fenner et al. [2]. They establish exactly which grids are 2-colourable and 3-colourable. In addition, they classified almost all grids by their 4-colourability. At the time of writing, it is unknown whether 12×21, 17×17, 17×18, 18×18 (and by rotation, 21×12 and 18×17) are 4-colourable or not. The paper is very technical and introduces a variety of concepts and techniques to prove the k-colourability of grids. Among these are rectangle-free subsets. A rectangle-free subset of an N×M grid is a subset of cells of the grid that do not contain a rectangle. In other words, if all the cells of a rectangle-free subset were assigned a colour, and the other cells were left unassigned, then the grid would have no monochromatic rectangles. By the pigeon-hole principle, if a grid is k-colourable then it has a rectangle-free subset whose cardinality is greater than or equal to $\lceil \frac{NM}{k} \rceil$ where $\lceil x \rceil$ is the smallest integer not less than x. The Rectangle Free Conjecture is a strong version of the converse of this. It states that if there exists a rectangle-free subset of size $\lceil \frac{NM}{k} \rceil$ then the N×M grid is k-colourable. This is unproven, but they observe that if the conjecture were true, then the three grids in question would be 4-colourable as they provide rectangle-free subsets that match the criteria for 12×21, 17×17 and 18×17. Furthermore, the rectangle-free subset they find for 17×17 has size 74, one greater than $\lceil \frac{17^2}{4} \rceil = 73$, which suggests that a 4-colouring for 17×17 is more likely to exist than one for 17×18, 18×18 or 12×21.

Kupin [6] examines 17×17 from a different perspective. She considers the possible placements of the most frequently appearing colour in a potential 4-colouring, and establishes that it must appear no more than 74 times in any potential 4-colouring of 17×17. By the pigeonhole principle, this colour must appear at least 73 times, so her results leave only two possibilties for how many times this colour must occur. Furthermore, she proves that no colour can occur more than five times in any row or column of a 4-colouring of 17×17. This is a useful result in terms of attempting to solve this problem with constraint programming.

Håkan Kjellerstrand has written a few simple constraint programs in MiniZinc for the grid colouring problem [7]. His programs are very basic, only containing the constraint which disallows monochromatic rectangles and a constraint which fixes the top-left and bottom-right corners to take the values 1 and 2 respectively. While this is not a serious attempt at solving the problem, his models, particularly the problem constraints, could be useful and are examined in greater detail in the next chapter.

3 Modelling The Grid-colouring Problem

In Section 2.1, 17×17 was formally defined as a CSP triple (X, D, C). This problem specification is not ideal for a constraint programming language. A concrete model is needed so that this problem can be expressed in a language such as MiniZinc.

Fortunately, the problem specification itself gives rise to a natural and obvious way to concretely model the problem, one which Kjellerstrand used in his attempt to solve the problem [7]. The goal is to fill an $N\times M$ grid, which can naturally be represented as a matrix, or a two-dimensional array, with N rows and M columns. For a 17×17 grid, this will be a two-dimensional array of 289 decision variables, the correct amount of variables required as specified by the original CSP. This is known as a matrix model. Furthermore, instead of using explicit colours as the values in the domain, the integers $\{1 \dots c\}$ will be used to represent each colour, where c is the number of colours to be used. The words 'values' and 'colours' will be used interchangeably when referring to the constraint satisfaction models. This model can be easily constructed for grids of any size. The constraints will be expressed in terms of the array elements, as demonstrated in the remainder of this chapter.

3.1 The Rectangle Constraint

For a grid G of size $n\times m$, I define a rectangle to be a 4-tuple of pairs:

$$((r_1, c_1), (r_1, c_2), (r_2, c_1), (r_2, c_2))$$

where $1 \leq r_1 < r_2 \leq N$ and $1 \leq c_1 < c_2 \leq M$.
Then the set of all rectangles in G is:

$$\{((r_1, c_1), (r_1, c_2), (r_2, c_1), (r_2, c_2)) \mid 1 \leq r_1 < r_2 \leq N,\ 1 \leq c_1 < c_2 \leq M\}$$

The rectangle constraint restricts at least two elements, or corners, of each rectangle in G to have distinct values. This is the only constraint in the grid-colouring problem. Propagation will occur when three corners of a rectangle are assigned the same value x. Then the variables representing each corner will have only the value x in their domains, and x can be removed from the domain of the variable representing the fourth corner to make the constraint GAC.

This constraint is not trivial to write in MiniZinc, and there are many ways of achieving this. Four ways of implementing this constraint are analysed in this chapter.

3.2 Analysing the Corner constraint

The first constraint will compare pairs of corners in each rectangle to ensure that not all corners are equal. The MiniZinc code for this constraint is shown in Figure 3.1.

```
constraint
forall(r1 in 1..r-1, r2 in 2..r, c1 in 1..c-1,
  c2 in 2..c where r1<r2 /\ c1<c2) (
    not(grid[r1,c1]=grid[r1,c2] /\ grid[r1,c1]=grid[r2,c1]
      /\ grid[r1,c1]=grid[r2,c2])
);
```

Figure 3.1: One way of writing the corner constraint. r is the number of rows of $grid$, and c is the number of columns.

The four quantified variables declared in the *forall* statement are analogous to the variables given in the set of all rectangles defined at the start of the chapter.

The reader should note that the choice of pairs of corners that are linked together are almost completely arbitrary. Let v_1, v_2, v_3 and v_4 be the top left, top right, bottom left and bottom right corners respectively. Then the above MiniZinc code fragment contains a negated conjunction of the expressions $v_1 = v_2$, $v_2 = v_3$ and $v_3 = v_4$. With three such expressions, there are 16 distinct ways of linking all four corners together. In addition, there are further possibilities with conjunctions of four or more expressions.

Table 3.1 shows the performance of eight different corner constraints on grids of various sizes. A fixed search annotation has been used for all test runs: *varchoice* is *input_order*, which will first assign values to the variables in the top row, from left to right, then move down one row and repeat the process. This is called row-wise variable ordering. *constrainchoice* is *indomain_min*, which instructs the solver to assign to each variable the smallest value in its domain. This is called smallest value assignment.

all_six is the negated conjunction between all six pairs of corners. There is only one such constraint. The other constraints are expressed as a list of pairs, where (v_i, v_j) means that $v_i = v_j$ is in the negated conjunction. For example, $(v_1, v_2), (v_2, v_3), (v_3, v_4)$ represents the constraint displayed at the start of this section. The entries in the table show the number of choice points (CPs) each search explored followed by the time took, in seconds, to find the first solution. The number of choice points is the number of times the solver assigns a value to a variable with more than one value in its domain, therefore the number of choice points is a good indicator of how many times the solver backtracked.

The table shows that four of the constraints produce an identical number of choice points for each grid, which strongly suggests that these models explore the same search space. *all_six* is the slowest of these, suggesting that more constraints being checked will result in a longer search time, although the difference in time is negligible for grids of this size.

The other four constraints use considerably fewer choice points as the grid grows larger. There are two distinct choice patterns, one peaking at 193 choice points and the other peaking at 173 choice points.

Size	all_six	$(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_3, v_4)$	$(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4)$	$(v_1, v_3), (v_1, v_4), (v_2, v_3), (v_2, v_4)$
3×3	14 CPs/0.002s	14 CPs/0.001s	9 CPs/0.004s	14 CPs/0.002s
4×4	29 CPs/0.009s	29 CPs/0.006s	15 CPs/0.011s	29 CPs/0.013s
5×5	56 CPs/0.021s	56 CPs/0.014s	22 CPs/0.011s	56 CPs/0.018s
6×6	182 CPs/0.049s	182 CPs/0.017s	42 CPs/0.034s	182 CPs/0.045s
7×7	1430 CPs/0.153s	1430 CPs/0.044s	193 CPs/0.074s	1430 CPs/0.107s

Size	$(v_1, v_2), (v_2, v_3), (v_3, v_4)$	$(v_1, v_2), (v_1, v_3), (v_1, v_4)$	$(v_1, v_4), (v_2, v_4), (v_3, v_4)$	$(v_1, v_2), (v_1, v_3), (v_2, v_4)$
3×3	9 CPs/0.002s	9 CPs/0.001s	14 CPs/0.003s	9 CPs/0.002s
4×4	15 CPs/0.004s	15 CPs/0.007s	29 CPs/0.009s	15 CPs/0.004s
5×5	22 CPs/0.015s	22 CPs/0.018s	56 CPs/0.016s	22 CPs/0.009s
6×6	42 CPs/0.024s	40 CPs/0.037s	182 CPs/0.034s	40 CPs/0.030s
7×7	193 CPs/0.053s	173 CPs/0.072s	1430 CPs/0.080s	173 CPs/0.068s

Table 3.1: Results for various corner constraints.

There is a subtle observation to be made. The worst performing constraints have more than one conjunct expression involving the bottom-right corner, v_4. Conversely, the better performing constraints have only one conjunct expression involving v_4. This is due to the variable ordering the solver takes.

To demonstrate this, consider the solution for 3×3, which is the first solution output by all the constraints tested:

1	1	1
1	2	2
1	2	3

The solver will assign 1 to the first four elements of the grid under the given search strategy. No backtracking has occurred yet no complete rectangles have been formed with instantiated variables. At this point the solver has to choose what value to assign $grid[2, 2]$, which is the bottom-right corner of the first rectangle to be fully instantiated. For a constraint involving v_4 exactly once, such as the constraint in Figure 3.1, we have the constraint $\neg(v_1 = v_2 \wedge v_2 = v_3 \wedge v_3 = v_4)$ which is equivalent to $\neg(TRUE \wedge TRUE \wedge v_3 = v_4)$ at this stage in the search. This constraint must be satisfied, so the solver knows that $v_3 = v_4$ must evaluate to $FALSE$. Thus propagation will occur and the value 1 will be pruned from the domain of $grid[2, 2]$, giving it a smaller domain of $\{2, 3, 4\}$, and the value 2 is assigned to $grid[2, 2]$ which satisfies the constraint. On the other hand, a constraint with two or more expressions involving v_4 will not provide the solver with enough information to prune 1 from the domain of $grid[2, 2]$ to

enforce GAC, so it will assign it the value 1 and backtrack, resulting in an additional choice point. This will occur three more times in this example, in particular both 1 and 2 will be assigned to $grid[3,3]$ before the constraint is satisfied with the value 3.

Therefore, minimising the number of boolean expressions in the conjunct involving v_4 is much more beneficial than minimising the number of expressions in the conjunction. It should be noted that in larger grids, as more propagation and backtracking occurs, the same problem will occur for the other three corners, albeit not as frequently, due to the variable ordering. For the rest of the paper, the only corner constraint that will be used is $(v_1, v_2), (v_1, v_3), (v_2, v_4)$, denoted $corner$ since it exhibits the fewest choice points and produces the fastest search out of the two constraints with its choice point pattern.

There is another way to express this constraint. By applying one of De Morgan's Laws, namely $\neg(A \wedge B) = \neg A \vee \neg B$, this constraint can be written as a disjunction of inequalities, as seen in Figure 3.2.

```
constraint
forall(r1 in 1..r-1, r2 in 2..r, c1 in 1..c-1,
  c2 in 2..c where r1<r2 /\ c1<c2) (
    grid[r1,c1]!=grid[r1,c2] \/ grid[r1,c1]!=grid[r2,c1]
      \/ grid[r1,c1]!=grid[r2,c2])
);
```

Figure 3.2: An alternate way of writing the corner constraint in Figure 3.1

With the same search annotation used in the previous test, both forms of this constraint are compared. The alternate form is named $corner'$. The results are displayed in Table 3.2.

Size	Corner	Corner'
3×3	9 CPs/0.002s	9 CPs/0.002s
4×4	15 CPs/0.004s	15 CPs/0.003s
5×5	22 CPs/0.009s	22 CPs/0.006s
6×6	40 CPs/0.030s	40 CPs/0.012s
7×7	173 CPs/0.68s	173 CPs/0.039s
7×8	2,473,511 CPs /41.083s	2,473,511 CPs /57.962s

Table 3.2: Comparison of two versions of $corner$

.

The choice points are identical up to 7×8, which is to be expected since the constraints are essentially equivalent. However, $corner'$ is noticeably slower than $corner$ on the largest grid, so the original implementation will be used.

```
predicate extensional_conflict
 (array[int, int] of var int: table, array[int] of var int: x) =
   not exists(pattern in index_set_1of2(table)) (
     forall(j in index_set_2of2(table)) (x[j] = table[pattern, j])
);

constraint
  forall(r in Rows, r2 in 1..r-1, c in Columns, c2 in 1..c-1) (
    extensional_conflict(forbidden,
    [space[r,c], space[r2,c], space[r,c2], space[r2,c2]])
);
```

Figure 3.3: Rectangle constraint using extensional conflict written by Håkan Kjellerstrand.

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

Figure 3.4: The array forbidden

3.3 Three more ways of expressing the Rectangle constraint

Comparing pairs of corners is not the only way to implement this constraint. Håkan Kjellerstrand [7] has written three Minizinc models of his own to attempt to solve this problem. The first of these models is described below.

In Figure 3.3, Håkan defines a constraint called *extensional_conflict* (although he names it *existential_conflict* in his blog) which takes a two-dimensional array of integers, *table* and a one-dimensional array of integers, x. The constraint is true if there does not exist a row in *table* that is equivalent to x. This constraint is used to compare the fixed array *forbidden*, shown in Figure 3.4, and the one-dimensional array $[space[r, c], space[r_2, c], space[r, c_2], space[r_2, c_2]]$ representing all the possible rectangles in the grid. *extensional_conflict* ensures that the rectangles do not match with the rows of *forbidden*. That is to say, all the rectangles do not have corners of the same colour. This constraint shall be referred to as EC.

Håkan's second model is an alternate version of the above program which defines *extensional_conflict* in a different way but is otherwise equivalent. His third model uses a global constraint called $global\_cardinality$. However, it uses an old version of the constraint that takes two arguments. I have modified the program to use a newer version which takes three arguments.

$global\_cardinality$ takes three arrays, $target$, $elements$ and $counts$, as arguments. It ensures that $elements[i]$ occurs in $target$ exactly $counts[i]$ times. The **let** construct is used to

```
constraint
  forall(r1 in 1..r-1, r2 in 2..r, c1 in 1..c-1, c2 in 2..c
         where r1<r2 /\ c1<c2) (
  let {array[1..4] of var 0..4: gcc} in
    global_cardinality
       ([grid[r1,c1], grid[r1,c2], grid[r2,c1], grid[r2,c2]],
        [1,2,3,4],gcc
    ) /\
   forall(i in 1..4) (gcc[i]<4)
);
```

Figure 3.5: Modified version of the rectangle constraint using global cardinality written by Håkan Kjellerstrand.

```
constraint
 forall(i in Colours, r1 in 1..r-1, r2 in 2..r,
   c1 in 1..c-1, c2 in 2..c where r1<r2 /\ c1<c2)(
    bool2int(grid[r1,c1]=i)+bool2int(grid[r1,c2]=i)+
    bool2int(grid[r2,c1]=i)+bool2int(grid[r2,c2]=i)<4
);
```

Figure 3.6: The Boolsum constraint.

declare a local array of variables called gcc, which is the third argument to $global\_cardinality$. This array is only visible within this constraint. Using the $forall$ operator, the constraint is applied to all rectangles in $grid$. If all four elements of the first array take the same value, then one of the elements of gcc will be 4 (and the rest will be 0). Therefore if $gcc[i] = 4$ for any i, then the rectangle constraint is unsatisfied, hence $gcc[i] < 4$ for all i, which is the second part of the constraint in Figure 3.5. This constraint shall be referred to as GC.

Another way of implementing the rectangle constraint is a constraint called $boolsum$, shown in Figure 3.6. The operator *bool2int* converts a Boolean expression into an integer in the standard way, i.e. FALSE = 0, TRUE = 1. For each i in the set $\{1 \dots c\}$, the sum of $bool2int(corner = i)$ for all four corners of each rectangle is constrained to be strictly less than 4.

Using the same search strategy as the experiments in Section 3.2, EC, GC, $boolsum$, and $corner$ are compared. The grid sizes tested are the same as those in Table 3.1. A table of results is not included here because EC, GC and Boolsum have the same number of choice points as $corner1$ (see the column labelled $(c_1, c_2), (c_1, c_3), (c_1, c_4)$ in Table 3.1). This suggests that EC, GC and $boolsum$ explore the same search space and perform identical propagation to $corner$. The differences in runtime are negligible on grids up to 7×7. For a 7×8 grid, EC took 78.382 seconds to find a solution and $boolsum$ took 74.936 seconds, certainly not as fast as $corner$. Furthermore, GC took 615.054 seconds to find a solution for 7×8 (with approximately 2.5 million choice points) which is a lot slower than any of the other constraints.

$corner$ is still the best performing constraint, so this shall be the rectangle constraint I used for the rest of the project.

The goal of this project is to solve 17×17 and $corner$ alone cannot solve even 8×8 in a reasonable amount of time. More tools are needed to improve performance. One such tool is a common constraint programming technique called symmetry breaking, introduced in the next chapter.

4 Symmetry Breaking

Symmetry breaking is a very important part of constraint programming, and has undergone a lot of study in recent years [8]. Intuitively, given one solution of a CSP, additional solutions can be derived directly from that solution by transforming it in some way. Such transformations are called symmetries. These solutions are essentially equivalent, therefore it is unnecessary to consider those additional solutions during search. The goal is to remove as many equivalent solutions from the search space as possible without losing every solution in a class of symmetric solutions. This is called symmetry breaking.

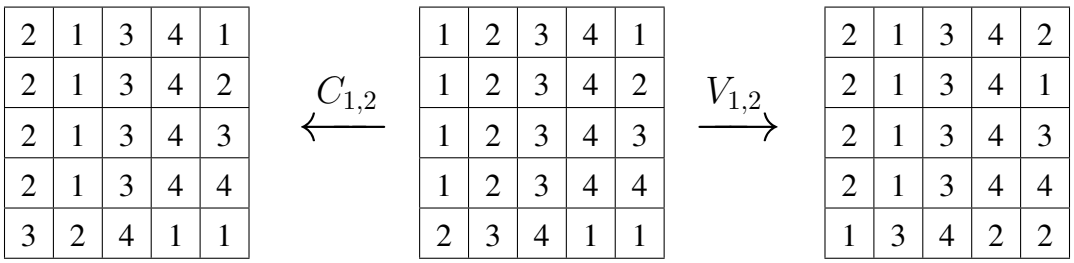

Figure 4.1: Three symmetrically equivalent solutions to 5×5

Figure 4.1 shows three 5×5 grids. The transformation $C_{1,2}$ has been applied to the middle grid to give the leftmost grid. This transformation swaps, or transposes, the first and second column of the grid. Similarly, the transformation $V_{1,2}$, which permutes the values 1 and 2, has been applied to the middle grid to give the grid on the right. All three grids are 4-colourings. It is clear that both of these transformations are invertible. In addition, both transformations will always map a solution into a solution and a non-solution into a non-solution.

I consider two kinds of symmetry: Variable symmetry and value symmetry. A variable (value) symmetry is a bijection on variables (values) that preserves solutions and non-solutions (solution-preserving) [9]. It is important to distinguish between the two, as symmetry breaking methods for variable symmetry and value symmetry differ [10]. In the above example, $C_{1,2}$ is a variable symmetry, as it will always swap the first two columns, regardless of the values their associated variables take. In contrast, $V_{1,2}$ is a value symmetry. It will permute the numbers in the grid, regardless of their positions. These symmetries can be composed to form symmetries that are neither variable nor value symmetries. However, breaking these symmetries separately will break some of the compositions as well. Since symmetries are bijections, each total assignment has an equivalence class of symmetric solutions called its symmetry class.

4.1 Symmetries in N×M

The symmetries in this CSP are not hard to identify. If an arbitrary colour-filled $N \times M$ grid contains a monochromatic rectangle, that is, a rectangle whose corners are the same colour, then it is clear that swapping any two rows or any two columns will not remove a monochromatic rectangle. Similarly, swapping any two rows or columns of a 4-colouring will not create a monochromatic rectangle. This idea can be extended. Columns and rows can be permuted indefinitely to create a new grid, and this transformation is solution-preserving. Clearly row and column permutations are invertible, so these transformations are variable symmetries.

Such transformations are called row and column symmetries, and they occur frequently in matrix models. This is a very important symmetry to break as there are $N!M! - 1$ possible compositions of row and column transpositions, excluding the identity transformation, for an $N \times M$ matrix. The sizes of symmetry classes will grow super-exponentially as the matrix size increases.

Another type of symmetry is the geometrical symmetries of a rectangle. These are rotations and reflections. If $N \neq M$, then there are only four symmetries: The identity transformation, a horizontal reflection, a vertical reflection, and a 180° rotation. However, these three symmetries can be expressed in terms of row and column transpositions. Figure 4.2 illustrates this for a horizontal reflection on a 6×4 grid. Similarly, a vertical reflection is equivalent to a sequence of column transpositions, and a 180° rotation is equivalent to a sequence of row transpositions and column transpositions.

1	1	2	2	3	1
3	1	4	4	4	2
2	2	2	3	4	2
2	1	3	4	1	4

$\xrightarrow{R_{1,4} \circ R_{2,3}}$

2	1	3	4	1	4
2	2	2	3	4	2
3	1	4	4	4	2
1	1	2	2	3	1

Figure 4.2: A 4×6 grid horizontally reflected by two row transpositions.

For square grids, however, there are four additional symmetries, namely a 90° rotation, a 270° rotation, and the two diagonal reflections. These transformations are solution-preserving, bijective, and none can be expressed as a sequence of row and column transpositions. There are only four such symmetries, regardless of the size of the grid, so breaking these symmetries isn't as important as breaking the row and column symmetries.

The final symmetry in N×M is the value symmetry. The values, or colours, are indistinguishable. That is to say, given a 4-colouring of an N×M grid, the colours can be permuted abritrarily, and the resulting grid will still be a 4-colouring. This symmetry was introduced when the problem was modelled, during which the colours were given integer values. For the 4-colouring problem, there are $4!$ value symmetries, including the identity, for a grid of any size.

4.2 Breaking the Symmetry

There are three common ways to break symmetry [8]. The first method is to remodel the problem. Modelling the problem in a different way can remove symmetries present in the original model and introduce different symmetries that may be easier to break. An alternate model is considered in Chapter 6.

The second method is dynamic symmetry breaking. This method breaks symmetry during search by preventing the solver from visiting a symmetrically equivalent solution to one already explored. Several methods have been developed to break symmetry dynamically [8]. However, MiniZinc does not possess the necessary tools to implement such techniques. In addition, this method may not be feasible for large matrix models. Smith and Gent [11] have applied a dynamic symmetry breaking technique called Symmetry Breaking During Search (SBDS) to the row and column symmetries of a matrix model. This technique adds constraints after each variable assignment during the search process. Due to the large amount of row and column symmetries present in matrix models, they found it is infeasible to break all row and column symmetries with this method except for very small matrices. They show that applying a suitable subset of the SBDS constraints is very effective, although they only tested them on 3×3 and 4×4 matrices. It is certainly possible that this application of SBDS would be effective for larger matrices, and the 4-colouring problem, but this is not an avenue explored in this project.

The third method is adding additional constraints to the model in order to break the symmetry before search. These are called symmetry breaking constraints. This is a well studied area for matrix models [9] [11] [12], and it is the method that will be discussed for the remainder of this chapter.

4.3 Symmetry Breaking Constraints

Lexicographic ordering of rows and columns is a commonly used symmetry breaking constraint to break row and column symmetry. This constraint will restrict the rows and columns of a two-dimensional matrix, treated as vectors, to be in lexicographic (non-strict) increasing order from left to right and from top to bottom respectively. Placing the rows and columns in decreasing order is called anti-lexicographic ordering. Flener et al. [9] prove that each symmetry class of a matrix model with row and column symmetry contains an element whose rows and columns are in lexicographical order. Therefore these constraints will not remove any solutions from the search space. Lexicographically ordering the rows and columns will not break all possible row and column permutations, but their experiments show that these constraints break enough symmetry to be effective in practice. I used the strict version of this constraint, *strict_lex2*, since for 5×5 grids and larger, if two rows or columns are the same, then the grid will have a monochromatic rectangle as each row and column must contain a repeated value. By abuse of notation, I will refer to the strict version as *lex2*.

Frisch, Jefferson and Miguel [12] propose a constraint *allperm* to be used in conjunction with *lex2*. *allperm* constrains the first row to be lexicographically less than or equal to all permutations of every other row. Their experiments show that the conjunction of *lex2* and

allperm produces faster search and fewer backtracks than just *lex2*.

Kiziltan and Smith [13] discuss lexicographically ordering columns, *col-lex*, in conjunction with different orderings for the rows. They consider ordering the rows by the sums of their values, or by their multisets, and breaking ties with lexicographic ordering. Their results show that *col-lex* with row-multiset ordering and lexicographical tie breaking is the most effective, but their experiments are limited to 3×3 and 4×4 matrices. It is unknown whether these constraints will perform as well on the larger matrices that this problem requires. Multiset ordering is very closely related to *allperm*. In fact, anti-multiset ordering has been used by Frisch et al. as an alternate characterisation for *allperm* [12]. Placing the rows in non-decreasing order of their sums, from top to bottom, is inconsistent with row lexicographic ordering. If the first row of an $N\times N$ matrix has a 1 in its leftmost position and $N-1$ 4s everywhere else, and the second row is filled with 2s, then the first row is lexicographically smaller than the second row but has a larger sum. Indeed, the same applies to placing the rows in non-increasing order of their sums. To see this, swap the 2s and 4s in the previous example.

To the best of my knowledge, no serious research has been conducted on rotational and reflectional symmetry breaking. This is probably due to the fact that there are only eight such symmetries, a small amount compared to the other symmetries of the problem.

The Lex-Leader Method [8] is a general technique for generating symmetry breaking constraints. One member of each symmetry class is chosen to be the canonical element. Symmetry breaking constraints are added that are satisfied by the canonical element and, in the best case, by no other elements. This can be used to generate symmetry breaking constraints if the four corner variables of a square grid are considered. Recall that for a rectangular grid, all the rotational and reflectional symmetries are column and row symmetries.

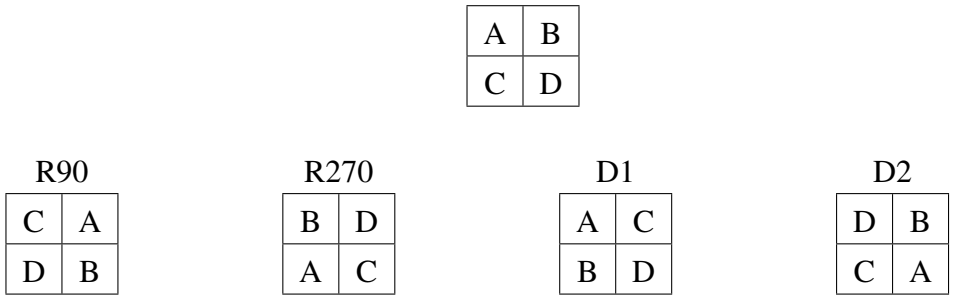

Figure 4.3: Lex-Leader on rotational and reflectional symmetries. R90 and R270 are rotations through 90° and 270° clockwise respectively. D1 is the reflection through the top-left to bottom-right diagonal. D2 is the reflection through the top-right to bottom-left diagonal.

Figure 4.3 depicts the Lex-Leader method being used to generate four lexicographical ordering constraints corresponding to the rotations and reflections of the canonical element, a square matrix with corners ABCD. Some simplications can be made. Constraint (3) reduces to B≤C since A=A trivially. Similarly, (4) reduces to A≤D. For (1), we have A≤C or A=C and B≤A. If A=C, then B≤A is satisfied by (3), so we now have A≤C or A=C and C≤D. If A=C then A≤D is satisfied by (4), so (1) reduces to A≤C. (2) reduces to A≤B in a similar manner. Note that A≤B and A≤C are consistent with the lexicographic ordering constraints.

The final symmetry to break is the value symmetry. Interchangeable values are a common feature of CSPs. Law and Lee [14] discuss value precedence, a set of constraints that breaks value symmetry in any CSP with indistinguishable values. These constraints are defined for sequences of integer variables and sequences of sets of integers variables. The value x has precedence over the value y, or $x \prec y$, in an integer sequence $<v_1, \ldots, v_n>$ if x occurs in the sequence before y. That is to say, if there exists j such that $v_j = y$, then there must also exist $i < j$ with $v_i = x$. For a sequence of sets, $x \prec y$ if any set containing y but not x occurs after a set containing x but not y. Both forms of value precedence are not useful for the 4-colouring problem for suitably large grids, where it can be assumed that all four colours appear at least once in every row or column. If we consider the matrix as a sequence of integers formed by appending the rows together from top to bottom, then placing the columns in lexicographic increasing order entails the value precedence constraints $1 \prec 2$, $2 \prec 3$ and $3 \prec 4$. Meanwhile if we consider each row, say, as a set of integers and order the sets from top to bottom, then the above value precedence constraints for sequences of sets will not remove any grids from the search space since there will be no sets containing one value and not another.

Flener et al. [9] briefly discuss a method to break value symmetry in matrix models. They propose creating a new auxiliary matrix of 0/1 variables with a dimension one greater than the original matrix, in order to replace the value symmetries with variable symmetries which can be broken with known techniques such as lexicographic ordering. For $N \times M$, this involves creating a $N \times M \times 4$ array A of variables with domain $\{0, 1\}$, with $A[i, j, c] = 1 \Leftrightarrow grid[i, j] = c$. Then the value symmetry can be broken by flattening each of the four N-M planes (one way is to flatten the planes row-wise from top to bottom) and lexicographically ordering them. This is equivalent to value predence. For example, consider the constraint $A[i, j, 1]$ is lexicographically greater than $A[i, j, 2]$. This means the first 1 in $A[i, j, 1]$ must occur before the first 1 in $A[i, j, 2]$. Note that the channelling constraints prevent them from both having a 1 in the same position. This happens if and only if 1 appears before 2 in $grid$, or $1 \prec 2$.

Another possible symmetry breaking constraint is *firstrow*. Define $\#_r x$ to be the number of times the value x occurs in row r. Then *firstrow* is the constraint $\#_1 1 \geq \#_1 2 \geq \#_1 3 \geq \#_1 4$. If $\#_1 x = \#_1 y$ for some values x and y, then the constraint $\#_2 x \geq \#_2 y$ is added. Every symmetry class under the value symmetries contains at least one grid satisfying this constraint, since the values are indistinguishable and can hence be relabelled to fit the constraint. Thus *firstrow* does not lose any solutions. This constraint is also consistent with *lex2*. For any arbitrary 4-colouring of an $N \times M$ grid, there exists a pair (possibly more than one) (i, v) such that $\#_i v = \max\{\#_r x \mid 1 \leq r \leq N, 1 \leq x \leq 4\}$. Ties are broken by considering the rows with the largest frequency for the second most frequent values. Further ties are broken by considering the third most frequent values. If this tie is not broken then the rows concerned will have exactly the same value frequencies, in which case the row chosen from these does not matter. If $r \neq 1$, then row r can be swapped with row 1, and the values can be permuted to satisfy *firstrow*. Also the columns and all other rows except the first can be permuted to lexicographically order the rows and columns. By construction, the first row will be the lexicographically least row if arranged in increasing order, which can always be done since all the columns can be permuted. If the variables are labelled row-wise, then this constraint will propagate quickly as only the first two rows have to be filled for this constraint to be evaluated.

```
array[Rows,Colours] of var 0..c: row_freq;

constraint forall (i in Rows) (
  global_cardinality([grid[i,j] | j in Columns], [1,2,3,4],
    [row_freq[i,1],row_freq[i,2],row_freq[i,3],row_freq[i,4]])
);
```

Figure 4.4: The array *row_freq* and its associated channelling constraints. The integer c is the number of columns.

4.4 Implementing the Symmetry Breaking Constraints in MiniZinc

Several of the symmetry breaking constraints mentioned in Section 4.2 are applied to rows of the grid, so I created an auxiliary array called *row_freq*. This array keeps track of the frequencies of each value in each row via channelling constraints. That is, $row\_freq[r, c] = \#_r c$. The array declaration and channelling constraints are shown in Figure 4.4. The channelling constraints use the global constraint $global\_cardinality$, first met in Section 3.3.

The Minizinc code for some of the symmetry breaking constraints detailed in the previous section are shown in Figure 4.5.

The first symmetry breaking constraint, *lex2*, is another MiniZinc global constraint. MiniZinc provides both a non-strict (*lex2*) and strict version (*strict_lex2*) of this constraint. As mentioned previously, I used the strict version.

allperm is easily implemented by using the array *row_freq*. A grid satisfies *allperm* if and only if its first row is lexicographically less than or equal to all permutations of every other row, if and only if its first row is lexicographically less than or equal to the lexicographically smallest permutation of every other row. From the definition of lexicographic ordering, this happens if and only if $(\#_1 1 > \#_k 1) \vee (\#_1 1 = \#_k 1 \wedge \#_1 2 > \#_k 2) \vee \ldots \quad \forall 2 \leq k \leq r$, where r is the number of rows. Note that this is, in fact, anti-multiset ordering. This is also equivalent to constraining the first row of *row_freq* to be lexicographically less than or equal to all other rows of *row_freq*. The global constraint $lex\_greatereq$ is used to implement this.

col-lex+row-sum is expressed using the global constraint lex_less and the aggregation operator sum, which computes the sum of an expression over a range of values, which is in this case the set Columns = $\{1, \ldots, c\}$.

Like *allperm*, *firstrow* is easily implemented with the use of *row_freq*, by ensuring the first row of *row_freq* is in non-increasing order. An if statement cannot be used to break ties since MiniZinc does not allow decision variables in the conditional expression. Instead, the ties are broken using an implies statement.

Table 4.1 shows results for the three row and column symmetry breaking constraints combined with the value symmetry breaking constraint, *firstrow* and the rotation/reflection symmetry breaking constraints. Blank entries indicate that no solution was output within 10 minutes, a convention that shall be used throughout this report. The search strategy used here is the

```
% lex2
constraint strict_lex2(grid);

% allperm
constraint forall (i in 2..r) (
  lex_greatereq([row_freq[1,c] | c in Colours],
                [row_freq[i,c] | c in Colours])
);

% col-lex + row-sum
constraint
 forall (j in 1..c-1)
  (lex_less([grid[i,j] | i in Rows], [grid[i,j+1] | i in Rows]))
 /\
 forall (i in 1..r-1)
  (sum (j in Columns) (grid[i,j]) <= sum (j in Columns) (grid[i+1,j])
);

% rotation/reflection constraints
constraint grid[1,1] <= grid[1,c];
constraint grid[1,1] <= grid[r,1];
constraint grid[1,1] <= grid[r,c];
constraint grid[1,c] <= grid[r,1];

% firstrow
constraint forall(c in 1..cols-1) (
 (row_freq[1,c] >= row_freq[1,c+1]) /\
  (row_freq[1,c] = row_freq[1,c+1] -> row_freq[2,c] >= row_freq[2,c+1])
);
```

Figure 4.5: Symmetry Breaking Constraints in MiniZinc.

Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
6×6	34 CPs	34 CPs	56 CPs	40 CPs
7×7	82 CPs	82 CPs	405 CPs	173 CPs
8×8	248,393 CPs/ 10.169s	248,393 CPs/ 10.447s	1,270,325 CPs/ 46.446s	- -

Table 4.1: Results for models with symmetry breaking constraints.

same one used in Chapter 3. The variable ordering is certainly a sensible one for these models as there are many constraints on the rows of the grid, particularly on the first row. All solutions to 6×6 and 7×7 were found in less than a second. The corresponding results for the model in Chapter 3, *no-sym*, which has no symmetry breaking constraints, are included for comparison.

Both models with *lex2* used the same amount of choice points with a negligible difference in time. This suggests that *allperm* is not breaking any more symmetry than *lex2*. These two models also have less choice points than the model with no symmetry. Despite this, the choice points still appear to be growing exponentially, and the models were unable to find a solution for 9×9 within 10 minutes. *col-lex+row-sum* backtracks more than *no-sym* for 6×6 and 7×7, which is somewhat surprising, as the constraints should, and most likely do, reduce the size of the search space. The cause of these results are examined in greater detail in the next section.

4.5 Value Assignment

In the experiments carried out so far, a fixed search strategy was used. In particular, the smallest value in each domain was assigned to its variable. This resulted in the first solution found by the solver having a specific form. For example, the 6×6 and 7×7 solutions output by *lex2*, *lex2+allperm* and *no symmetry* under this search strategy are shown below.

1	1	1	1	1	1
1	2	2	2	2	2
1	2	3	3	3	3
1	2	3	4	4	4
1	3	4	2	3	4
1	3	4	2	4	3

1	1	1	1	1	1	1
1	2	2	2	2	2	2
1	2	3	3	3	3	3
1	2	3	4	4	4	4
2	3	4	1	2	3	4
2	3	4	1	3	4	2
2	3	4	1	4	3	2

Both of these solutions satisfy *lex2* and *allperm*. However, $sum(row\,4) > sum(row\,5)$ in both solutions, so neither of them satisfy *col-lex+row-sum*. This observation explains the results of the previous section. With no constraints other than the problem constraints, the solver will find this solution before any other solution. However, *col-lex+row-sum* will discard this solution since it does not satisfy *row-sum*, hence it will backtrack and look elsewhere in the search space to find the first solution that satisfies *row-sum*. Even though the search space has been reduced, due to this backtracking, it has to explore more of the search space than any of the other models. In contrast, these solutions satisfy *lex2* and *allperm*, and the corresponding models will find the solution faster than *no-sym* since the symmetry breaking constraints have reduced the search space.

Another observation is that these two solutions have a similar form, particularly in their first four rows. This is due to the smallest value assignment and row-wise variable selection. Each element in the first row and the first element in the second row have the value 1. The remaining elements on the second row will have the value 2 as 1 is removed from their domains by prop-

Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
6×6	121 CPs	121 CPs	42 CPs	36 CPs
7×7	66 CPs	69 CPs	61 CPs	54 CPs
8×8	578 CPs	479 CPs	67 CPs	75 CPs
9×9	130 CPs	130 CPs	-	111 CPs
10×10	769 CPs	18015 CPs/1.351s	-	121 CPs
11×11	-	-	-	220 CPs

Table 4.2: Results for symmetry breaking constraints with random value assignment. All solutions were found under a second except for 10×10 with *lex2+allperm*.

agation on the rectangle constraint. This pattern will continue for the third and fourth rows. Hence this search strategy guides the solver through a specific path in the search space, and for larger grids, a solution of this form may not even exist, which will result in backtracking and more search being performed. Indeed the solutions with this form are a very small subset of the total number of solutions, and there could be many other solutions "higher up" in the search space.

An alternative is to use a non-deterministic value assignment strategy. As far as I am aware, MiniZinc has only one such strategy, $indomain\_random$, which chooses a random value from the domain of each variable. The reader should note that the solver does not generate different random values on each program. I cannot find any documentation on the implementation of the random number generator, but it will generate the same values on consecutive runs of the same model, so the results presented are certainly a good indicator of how the model performs as the grid size increases.

Table 4.2 contains the results for grids of various sizes. The only modification to the model is the replacement of $indomain\_min$ with $indomain\_random$. There is a noticeable gain in search efficiency. Three of the models were able to find a solution for 10×10, and *no-sym* found a solution for 11×11 in under a second, outperforming the models with symmetry breaking. This happens because, with random value assignment, the first solution found by a model with no symmetry breaking does not satisfy any of the symmetry breaking constraints. The models with symmetry breaking won't accept this solution and have to search further to find a different solution. The amount of extra search required varies for grids of different sizes, 8×8 in particular takes far more backtracks to find for the *lex2* models than it does for 6×6, 7×7 or 9×9. While *col-lex+row-sum* performs better than the other symmetry breaking models on the smaller grids, it does not find any solutions for 9×9 or larger within 10 minutes.

Despite the random value ordering, none of the models can find a solution for 12×12 within 10 minutes. Symmetry breaking has improved the search efficiency but the current model is still a long way from reaching 17×17.

5 Further Search Space Reduction

5.1 Discussion

The search space for this problem is huge, and grows exponentially as the grid size increases. While symmetry breaking does prune the search space, it isn't effective enough to find a solution for any grids larger than 11×11 in a reasonable time. One property of symmetry breaking constraints is that they do not remove entire symmetry classes of solutions from the search space. In addition, the models used do not break all symmetries, so there will still be many symmetric solutions in the search space. Since the goal is to find only one solution, there is no danger in removing symmetry classes from the search space as long as it is done in a controlled and sensible way.

As demonstrated in Section 4.5, the constraints and search strategy guide the solver towards a very specific part of the search space, one that does not necessarily contain a lot of solutions. It would certainly be beneficial if the solver could be guided towards a solution-rich area of the search space. Therefore the problem needs to be examined in more detail in order to find properties that are shared by many solutions, including large grids such as 17×17 which will probably have very few non-symmetric solutions, if any at all.

It has been established that, in general, searching with random value assignment finds a solution more efficiently than smallest value assignment. The first solution found with smallest value assignment will tend to have more 1s than any other value, especially on the first row, and less 4s than any other value. On the other hand, random value assignment distributes the values more evenly. It is not a stretch, then, to assume that there are more solutions with a roughly even distribution of values than solutions with biased distributions since random value assignment exhibits better performance.

Kupin [6] arrives at a related conclusion for 17×17. She considers how many cells of the most frequently occuring colour C could fit in each column without creating a monochromatic rectangle by examining the pairs of rows covered by C in each column. A colour cannot occur in the same two rows in two distinct columns, since that would form a monochromatic rectangle. Using this fact, she proves that C can occur no more than five times and no less than four times in any column, which is a very even distribution.

Considering the number of pairs a colour takes in a column (or row) is a nice way of looking at the problem, and can be extended to grids of arbitary size. For example, a 12×12 grid has $\binom{12}{2} = 66$ pairs available for each colour, in both the rows and the columns. Therefore all four colours cannot cover more than 264 pairs between them. If the first column contains all 1s, then all 66 pairs have been taken by the 1s, and no pairs have been taken by the other three values, so 66 pairs are covered in total. If the first column contains six 1s and six 2s, then $\binom{6}{2} + \binom{6}{2} = 30$ pairs are covered between them. If the first column contains three of each

Smallest value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
9×9	743 CPS/0.165s	743 CPs/0.204s	1151 CPs/0.179s	2879 CPs/0.244s
10×10	43,693 CPs/3.196s	43,693 CPs/2.807s	26,329 CPs/2.623s	230,427 CPs/12.885s
11×11	121 CPs/0.220s	121 CPs/0.233s	91 CPs/0.200s	121 CPs/0.130s
12×12	112 CPs/0.265s	112 CPs/0.288s	109 CPs/0.256s	112 CPs/0.253s
Random value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
9×9	100 CPs/0.118s	91 CPs/0.143s	6673 CPs/0.535	60 CPs/0.095s
10×10	552,607 CPs/25.264s	117 CPs/0.179s	-	87 CPs/0.129s
11×11	4829 CPs/0.455s	7958 CPs/0.595s	-	132 CPs/0.181s
12×12	203 CPs/0.279s	68,859 CPs/4.015s	-	153 CPs/0.241s
13×13	541,490 CPs/53.435s	-	-	182 CPs/0.307s

Table 5.1: Results for models with row and column frequency bounds.

colour, then only $4\binom{3}{2} = 9$ pairs are covered in total. The most even distribution covers the least amount of pairs, and no other distribution of colours on the first row will cover fewer pairs. Conversely, if a row or column is filled with 1s, then a quarter of the total number of pairs have been covered in one twelfth of the cells, which restricts the distribution of colours in the remaining rows or columns far more than an evenly distributed row or column. This supports the conjecture that evenly distributed solutions occur more frequently than other solutions.

5.2 Experimenting with row and column value frequency bounds

From the discussion in the previous section, evenly distributing the colours among rows and columns will, in theory, guide the solver to an area in the search space with many solutions. This can be achieved by setting bounds for the amount of times each colour will appear in a row or column, called row (column) value frequency bounds. This is straight-forward to implement in MiniZinc. The model already has the auxiliary array *row_freq* which contains the frequencies of each colour on each row. Applying bounds on this is achieved by simply changing the domain of *row_freq* appropriately. I created a similar auxiliary array called col_freq which serves the same purpose for each column.

For an $N \times N$ grid, every colour should appear in each row and column approximately $\frac{N}{4}$ times. A good starting point is to restrict the domains of variables in *row_freq* and col_freq to $\{\lceil \frac{N}{4} \rceil - k, \ldots, \lceil \frac{N}{4} \rceil, \ldots, \lceil \frac{N}{4} \rceil + k\}$, where $\lceil x \rceil$ is the smallest integer not less than x. These domains will give balanced rows and columns while not completely restricting the row and column frequencies. For grids of size 12×12 and less, k is 1, and for any larger grids k is 2.

These bounds give some breathing space for the value frequencies but don't allow for solutions that are highly unbalanced.

The results for the MiniZinc models with the bounds added are given in Table 5.1. They find solutions for 12×12 and 13×13 in under a second, a clear improvement over the models without any row and column frequency bounds. The number of choice points are quite erratic, sometimes increasing or decreasing substantially after the grid size was increased. In particular, for smallest value assignment, it took far more choice points to find a solution for 10×10 than it did for any other grid, including larger ones. The models showed similar performance for11×11 and 12×12 with smallest value assignment, but with random assignment, *no-sym* performed a lot better than the other models.

The solutions output by the symmetry breaking models had the maximum number of 1s in the first row and column, due to the smallest value assignment. Surprisingly, this also occurred in most of the random value assignment solutions. In these cases the first row and column are as least balanced as possible. To rectify this, I modified the model by fixing the value frequencies of the first row and column. All other rows and columns retain the same bounds. For most grids, the first row and column won't have an even distribution of colours. In order to maintain consistency with the symmetry breaking constraints, the smaller the value, the more times it will appear. This is detailed in the MiniZinc code which fixes the value frequencies of the first row, shown in Figure 5.1.

```
int: q = c div 4;
constraint ((c mod 4) = 0)
  -> forall(c in Colours) (row_freq[1,c] = q);
constraint ((c mod 4) = 1)
  -> row_freq[1,1] = q+1 /\ forall(c in 2..cols) (row_freq[1,c] = q);
constraint ((c mod 4) = 2)
  -> row_freq[1,1] = q+1 /\ row_freq[1,2] = q+1 /\
     row_freq[1,3] = q /\ row_freq[1,4] = q;
constraint ((c mod 4) = 3)
   -> forall(c in 1..cols-1) (row_freq[1,c] = q+1) /\ col_freq[1,4] = q;
```

Figure 5.1: MiniZinc code to fix the value frequency of the first row.

First I declared a parameter $q = \lfloor \frac{c}{2} \rfloor$. Note that for square grids, $q = \lfloor \frac{r}{2} \rfloor$ since $c = r$, where c and r are the number of columns and rows respectively. div is an arithmetic operator, where $x\ div\ y$ returns the greatest integer no larger than $\frac{x}{y}$. The row frequency constraints are split into four cases, depending on the value of c mod 4. In the first case, c divides 4 evenly, so all four colours can be distributed evenly. In the second case, there is one extra spot, so 1 will occur once more than all the other values. In the third case, there are two extra spots, and as bias is given to the smaller values, 1 and 2 will both appear one more time than 3 and 4. Finally, when c mod 4 = 3, the value 4 will appear one less time than all the other values. The code that fixes the frequencies of the first column is analogous.

In addition, the variable ordering has been modified. The first row will be assigned values from left to right, then the first column from top to bottom, both under smallest value assign-

Smallest value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
11×11	136,540 CPs/9.240s	137,172 CPs/8.918s	202,537 CPs/16.649s	19,111 CPs/1.341s
12×12	846,583 CPs/65.901s	8,869 CPs/2.269s	15,381 CPs/1.160s	115 CPs/0.245s
13×13	490 CPs/0.412s	1,121 CPs/0.526s	771 CPs/0.407s	-
14×14	860,167 CPs/ 134.146s	4,327 CPs/0.999s	44,184 CPs/8.502s	-
Random value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
11×11	122 CPs/0.224s	121 CPs/0.248s	309,157 CPs/23.248s	121 CPs/0.120s
12×12	189 CPs/0.275s	147 CPs/0.282s	239 CPs/0.262s	161 CPs/0.230s
13×13	2,476 CPs/0.530s	2,356 CPs/0.569s	-	291 CPs/0.335s
14×14	-	3,271,273 CPs/ 428.169s	-	-
15×15	373,091 CPs/64.302s	-	-	-

Table 5.2: Results for the models with fixed value frequencies of the first row and column.

ment, followed by the $N-1 \times N-1$ subgrid obtained by removing the first row and column. The subgrid has the usual variable ordering, that is, across each row from left to right, with the rows being chosen in a top-down fashion. This variable ordering is used for the rest of the models in this chapter. The results for the modified model are given in Table 5.2. While *no-sym* performs badly, the symmetry breaking models in some cases find solutions for 14×14 and in one case 15×15.

Fixing the first row and column showed potential, so I extended this by fixing the row and column bounds of the first $\lfloor \frac{N}{2} \rfloor$ rows and columns. The whole grid should have a roughly balanced distribution, and placing the bias on the smaller values for approximately half the

Smallest value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
11×11	18,700 CPs/1.494s	12,868 CPs/1.102s	11,361 CPs/1.062s	-
Random value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
11×11	108 CPs/0.228s	112 CPs/0.233	122 CPs/0.204s	129 CPs/0.186s
12×12	158 CPs/0.279s	169 CPs/0.302s	519 CPs/0.279s	722 CPs/0.274s

Table 5.3: Results for the models with fixed value frequencies of the first $\lfloor \frac{N}{2} \rfloor$ rows and columns.

Smallest value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
11×11	216,174 CPs/13.740s	203,865 CPs/12.855s	130,621 CPs/10.412s	99 CPs/0.180s
12×12	13,736 CPs/1.038s	6,472 CPs/1.566s	11,853 CPs/0.132s	122 CPs/0.266s
13×13	3,048 CPs/0.540s	2,814 CPs/0.543s	2,310 CPs/0.506s	142 CPs/0.334s
14×14	1,744 CPs/0.638s	690 CPs/0.519s	2,694,166 CPs/467.521s	152 CPs /0.439s
15×15	921,499 CPs/175.871s	774,181 CPs/178.317s	2,551,701 CPs/444.432ss	-
Random value assignment				
Size	lex2	lex2+allperm	col-lex+row-sum	no-sym
11×11	109 CPs/0.241s	125 CPs/0.236s	174 CPs/0.217s	130 CPs/0.182s
12×12	127 CPs/0.287s	126 CPs/0.315s	122 CPs/0.258s	125 CPs/0.238s
13×13	142 CPs/0.357s	150 CPs/0.367s	1,995 CPs/0.525s	287 CPs/0.338s
14×14	106,738 CPs/13.206s	81,599 CPs/9.814s	104,294 CPs/11.724s	994 CPs/0.459s
15×15	373,091 CPs/64.302s	1,557,137 CPs/231.639s	-	-

Table 5.4: Results for the models with fixed value frequencies of the first $\lfloor \frac{N}{2} \rfloor$ rows and the first column.

rows and columns will likely produce unbalanced solutions, as the smaller values will appear more frequently across the whole grid. I programmed this in MiniZinc on a similar case-by-case basis to the previous model. For the row frequencies, $c \bmod 4 = 0$ is the simplest case, as each of the first $\lfloor \frac{N}{2} \rfloor$ rows will have an equal distribution of colours. When $c \bmod 4 = 1$, the first $\lfloor \frac{N}{2} \rfloor + 1$ rows will contain more 1s than any other colour, and the next $\lfloor \frac{N}{2} \rfloor$ rows will contain the most 2s. When $c \bmod 4 = 2$, the first $\lfloor \frac{N}{2} \rfloor$ rows will have an equal number of 1s and 2s, which will each occur exactly once more than 3 and 4. Finally, in the case $c \bmod 4 = 3$, the first $\lfloor \frac{N}{2} \rfloor + 1$ rows have the least 4s, and the next $\lfloor \frac{N}{2} \rfloor$ rows have the least 3s. The value frequencies for the first $\lfloor \frac{N}{2} \rfloor$ columns are analogous.

Table 5.3 shows that these models are weaker than any of the other models seen in this chapter, failing to find solutions for 13×13 in under 10 minutes. The models with smallest value assignments are particularly bad. The topmost $\lfloor \frac{N}{2} \rfloor$ rows and the leftmost $\lfloor \frac{N}{2} \rfloor$ columns have fixed frequencies, so the variables in the top-left quadrant of the grid are constrained twice by these constraints, which is probably the cause of the weak performance. To remedy this, I removed all the column frequency constraints except for the leftmost column. The results are in Table 5.4.

This modification is certainly an improvement as solutions for 15×15 are being found by most of the models, but no model could find a 16×16 solution in under 10 minutes. Indeed, fixing the frequencies of multiple rows and the first column is a step forward, so I tried extending this idea by fixing every row in the grid. Once again, to obtain the most balanced solutions, the row frequencies are restricted in such a way that all colours get approximately equal priority. The row frequencies are given in Figure 5.2. The frequencies have been fixed

Smallest value assignment			
Size	lex2	col-lex+row-sum	no-sym
11×11	147 CPs/0.251s	147 CPs/0.219s	88 CPs/0.187s
12×12	3,473 CPsS/0.494s	4,620 CPs/0.544s	118 CPs/0.241s
13×13	2,280 CPs/0.562s	2,337 CPs/0.606s	144 CPs/0.365s
14×14	1,575 CPs/0.664s	57,020 CPs/8.330s	73,414 CPs/10.589s
Random value assignment			
Size	lex2	col-lex+row-sum	no-sym
11×11	96 CPs/0.245s	121 CPs/0.235s	130 CPs/0.192s
12×12	126 CPs/0.278s	116 CPs/0.259s	125 CPs/0.241s
13×13	1,148 CPs/0.469s	181 CPs/0.350s	613 CPs/2.382s
14×14	-	206,242 CPs/25.928s	22,168 CPs/2.618s

Table 5.5: Results for the models with fixed value frequencies of all rows and the first column.

in this way in order to be consistent with the symmetry breaking constraints. In fact, *allperm* is implied by these frequencies since $\#_1 1 \geq \#_r 1$ for $r > 1$. Therefore the models *lex2* and *lex2+allperm* break exactly the same symmetries, and there is no need to test them separately. The results are presented in Table 5.5. Most of these models find a solution for 14×14 quite quickly, but they generally perform worse than those with only half the rows fixed. In addition, none of these models find a solution for 15×15 within 10 minutes.

5.3 Overview of Results

Overall, as conjectured at the start of this chapter, it is clear that restricting the number of appearances of each colour in every row and column generates more efficient search than previous models. Without these bounds, the model was unable to find a solution larger than 11×11 in reasonable time, but after fixing the frequencies of some rows and columns, solutions for 15×15 have been found. The model has become quite sophisticated, but it is not powerful enough to find solutions for any larger grids. Five different models have been built, with varying degrees of tightness on the value frequency bounds of the rows and columns, each tested with three of the row and column symmetry breaking constraints discussed in Chapter 4, the basic model with no symmetry breaking, and two different value assignment strategies. This section aims to summarise these results.

The difference in performance between the two value assignment strategies isn't as great as the difference seen for the models in Chapter 4. In most cases, both strategies were able to find solutions of the same size, although random value assignment was, in general, faster and backtracked less than smallest value assignment. This suggests that when the variables are more tightly constrained, a non-deterministic value assignment won't have as much significance as

	N mod 4 = 0	N mod 4 = 1	N mod 4 = 2	N mod 4 = 3
$\#_r1$	$\frac{N}{4}, 1 \leq r \leq N$	$\lceil \frac{N}{4} \rceil, 1 \leq r \leq \lceil \frac{N}{4} \rceil$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lceil \frac{N}{4} \rceil, 1 \leq r \leq \frac{N}{2}$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lfloor \frac{N}{4} \rfloor, \lceil \frac{3N}{4} \rceil + 1 \leq r \leq N$ $\lceil \frac{N}{4} \rceil$ otherwise
$\#_r2$	$\frac{N}{4}, 1 \leq r \leq N$	$\lceil \frac{N}{4} \rceil, \lceil \frac{N}{4} \rceil + 1 \leq r \leq \lceil \frac{N}{2} \rceil$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lceil \frac{N}{4} \rceil, 1 \leq r \leq \frac{N}{2}$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lfloor \frac{N}{4} \rfloor, \lceil \frac{N}{2} \rceil + 1 \leq r \leq \lceil \frac{3N}{4} \rceil$ $\lceil \frac{N}{4} \rceil$ otherwise
$\#_r3$	$\frac{N}{4}, 1 \leq r \leq N$	$\lceil \frac{N}{4} \rceil, \lceil \frac{N}{2} \rceil + 1 \leq r \leq \lceil \frac{3N}{4} \rceil$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lceil \frac{N}{4} \rceil, \frac{N}{2} + 1 \leq r \leq N$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lfloor \frac{N}{4} \rfloor, \lceil \frac{N}{4} \rceil + 1 \leq r \leq \lceil \frac{N}{2} \rceil$ $\lceil \frac{N}{4} \rceil$ otherwise
$\#_r4$	$\frac{N}{4}, 1 \leq r \leq N$	$\lceil \frac{N}{4} \rceil, \lceil \frac{3N}{4} \rceil + 1 \leq r \leq N$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lceil \frac{N}{4} \rceil, \frac{N}{2} + 1 \leq r \leq N$ $\lfloor \frac{N}{4} \rfloor$ otherwise	$\lfloor \frac{N}{4} \rfloor, \lceil \frac{N}{4} \rceil + 1 \leq r \leq N$ $\lceil \frac{N}{4} \rceil$ otherwise

Figure 5.2: Fixed row value frequencies for an $N \times N$ grid.

in a model with fewer constraints.

no-sym performed the best in general, but it was unable to find any 15×15 solutions within 10 minutes, something which all the symmetry breaking models did at least once. As the size of the grid grows larger, the number of symmetries increases, hence the symmetry breaking constraints will have a greater effect. *lex2* and *lex2+allperm* generally backtrack less than *col-lex+row-sum* and sometimes find solutions for larger grids. This is probably because *row-lex* is a stronger constraint than *row-sum*. The former not only considers the values the variables take, but also their position. The leftmost variables in each row have a greater significance than the other variables by definition of lexicographical ordering. Meanwhile, the sum of the values in each row clearly does not depend on the position of the values. *lex2+allperm* backtracked less than *lex2*, particularly for the models with smallest value assignment. This is not surprising as clearly *lex2+allperm* is stronger than *lex2*. Despite that, in the models which found a solution for 15×15, *lex2* found it in less time and with fewer backtracks than *lex2+allperm*. For 15×15, and probably for larger grids, it is harder to find a solution satisfying *lex2* and *allperm* than it is to find one just satisfying *lex2*.

Fixing the value frequencies of approximately half the rows and the first column produced the best results. Constraining the variables further by fixing the value frequencies of more rows or columns proved to be worse, particularly in the case where multiple columns and multiple rows had their value frequency bounds fixed. This suggests that for the 4-colouring problem, it is not hard to weaken search efficiency by applying too many constraints to the model, especially when trying to find solutions of larger grids. This is evidenced not only by the amount of rows and columns that have fixed value frequencies, but by the superior performance of *lex2* over *lex2+allperm* on 15×15 grids.

6 Extensions

Two extensions are discussed and tested. The first is the integration of the current model with a new model. The second is a series of modifications to the general grid-colouring model, specific for 16×16 and 17×17 grids.

6.1 The Pairs Matrix

When the problem was modelled in Chapter 3, the most natural model was chosen. While this model proved effective, it might be beneficial to look at the problem from a different perspective. Changing the model of the problem, or reformulation, is a frequently used technique in Constraint Programming [15]. In Section 5.1, the idea of searching for balanced solutions was motivated by considering pairs of elements in the grid. This idea can be taken a step further, and a new model can be built around the pairs occurring in the grid instead of the grid itself.

The driving force of this model is a new array of decision variables with domains $\{0, \ldots, 4\}$ called the pairs matrix, whose columns represent all the possible pairs that can be taken on each row of a grid. For an N×M grid, its associated pairs matrix will have $\binom{M}{2}$ columns. The extra domain value 0 is used to signify that there are two different values in the corresponding pair of the grid. An example is shown in Figure 6.1.

1	1	1	3
1	3	3	1
2	2	2	2
3	1	4	4

(1,2)	(1,3)	(1,4)	(2,3)	(2,4)	(3,4)
1	1	0	1	0	0
0	0	1	3	0	0
2	2	2	2	2	2
0	0	0	0	0	4

Figure 6.1: A 4×4 grid and its corresponding pairs matrix. The rows of the pairs correspond to the rows of the grid. The pairs indexing each column are the two columns of each possible pair of rows in the grid.

One advantage of this model is that the rectangle constraint can be expressed very easily. The grid is a 4-colouring if and only if no colour occupies the same pair in two different rows, so each colour must appear at most once in every column of the pairs matrix. This constraint propagates quickly as assigning any non-zero value to a variable will remove that value from the domain of all other variables in that column. In addition, the constraint introduces an upper bound on the total number of pairs each colour can take. Since each colour can appear in each column at most once, each colour can appear in the pairs matrix no more than $\binom{M}{2}$ times.

Other constraints are required to ensure that the pairs matrix correspondings to a legal grid. For example:

1. $\forall\, 1 \leq c \leq 4,\ pairs\_matrix[r,(x_1,x_2)] = c \Rightarrow \forall\, \{(y_1,y_2) \mid y_1 = x_1 \vee y_2 = x_2\},$ $pairs\_matrix[r,(y_1,y_2)] = 0 \vee pairs\_matrix[r,(y_1,y_2)] = c$

2. $\forall\, 1 \leq c \leq 4,\ pairs\_matrix[r,(x,y)] = c \wedge pairs\_matrix[r,(x,z)] = c \Rightarrow$ $pairs\_matrix[r,(y,z)] = c$

Constraint (1) prevents pairs matrix entries which imply that two different colours occupy the same cell in the grid. Constraint (2) states, in terms of the variables of the pairs matrix, that if the pairs (x, y) and (x, z), where x, y are z are distinct columns of the grid, are occupied by the same colour on the same row of the grid, then the pair (y, z) must also be occupied by the same colour on that row.

These constraints, particularly (1), are complex and quite awkward. Writing these constraints with the syntax and rules of MiniZinc was inelegant and although they were untested, I suspect they would have resulted in inefficient search and propagation. It is also impossible, using the pairs matrix alone, to write the constraint that each 0 in the pairs matrix implies there are two different values in the corresponding pair of the grid. These limitations motivate a model that combines the array $grid$ used in previous models in conjunction with the pairs matrix.

6.2 The Combined Model

This model will use both the pairs matrix and the array $grid$ from the original model. The pairs matrix is easily linked to $grid$ with channelling constraints. The channelling constraints are quite easy to express and they remove the necessity for the constraints discussed in the previous section, as they are implied by the channelling constraints.

Both the grid and the pairs matrix have row, column and value symmetry. The rows of the pairs matrix and the grid are in direct correspondence, so permuting the rows of the grid is equivale nt to permuting the same rows of the pairs matrix. The column symmetries aren't as straight-forward. Every column permutation of the grid corresponds to a column permutation of the pairs matrix. For example, swapping the first two columns of a grid is equivalent to swapping each pair of columns $(1, k), (2, k)$ of the corresponding pairs matrix. However, this is a one-to-one correspondence, and the pairs matrix has more columns than the grid, so applying an arbitrary column permutation to a legal pairs matrix may give an illegal pairs matrix. To see this, transposing the columns $(1, 4)$ and $(2, 3)$ in the pairs matrix of Figure 6.1 gives a new pairs matrix. The first three values in the first row of this new pairs matrix imply that 1 occupies the entire first row of the grid, yet the last three values in the first row are all 0, which implies the corresponding pairs have two different values, a clear contradiction. The value symmetry of the pairs matrix is not total. The values 1, 2, 3 and 4 can be permuted arbitrarily but swapping 0 with any other value in the pairs matrix is not a solution-preserving transformation.

The rows or columns of $grid$ being in lexicographic order does not imply that the rows or columns of the pairs matrix are in lexicographic order and vice versa. Figure 6.1 is a counter-example. The same applies for ordering the rows or columns by their sums or multisets. Therefore row and column symmetry breaking constraints cannot be applied to both $grid$ and the pairs matrix in the same model, but this is not a problem as fixing the rows and columns of one of the arrays will fix the rows and columns of the other.

Several methods of breaking row and column symmetry have been discussed and applied to $grid$ already. As shown with the constraints mentioned in the previous section, the values the variables of the pairs matrix take depend on each other due to the channelling constraints. This may have an impact on any constraints applied to the rows, since these dependencies occur between variables on the same row. It will likely have the greatest effect on *row-lex*, which places emphasis on the position of the variables as well as their values. Due to these dependencies, a symmetry breaking constraint which only considers the values and not their positions might have better performance on this model, so row multiset ordering, *row-set*, shall also be considered. *row-sum* will not be applied to this model as it was the worst performing constraint in the experiments of 5, and it is weaker than both *row-set* and *lex2*. Two rows that have the same multiset will always have the same sum, but two rows with the same sum could have different multisets. To be consistent with the value precedence constraints, the rows will be placed in lexicographic increasing order or in anti-multiset order. Since the pairs matrix only has partial column symmetry, no column symmetry breaking will be applied to it.

The value symmetry of the pairs matrix can be broken with value precedence constraints. A sensible and conventional choice is $1 \prec 2 \prec 3 \prec 4$, which is shorthand for $(1 \prec 2) \wedge (2 \prec 3) \wedge (3 \prec 4)$. The sequence of variables these constraints act on also needs to be taken into consideration. There are two natural choices. The first is the sequence obtained by taking the variables from the first row, left to right, and appending to that the variables from the second row, continuing in this way until the bottom row is reached. This means that if the values x and y both occur at least twice on the first row of $grid$, and $x \prec y$ on the pairs matrix, then the first occurence of x in $grid$ is before the first occurence of y due to the lexicographic ordering of the pairs that index the columns. I call this constraint *row-prec*. The second choice is taking the variables in a similar manner top to bottom from each of the columns, starting with the leftmost column, called *col-prec*. Assuming there is a row in $grid$ which starts with a pair, and that $x \prec y$, then this choice of sequence means that a row in $grid$ starting with xx will occur above a row starting with yy. These sequences could have a different effect on search efficiency, so both will be tested.

6.3 Implementation and Testing

Figure 6.2 shows some of the code for the combined model. An integer parameter num_pairs is used to store the number of possible pairs. Note that $\frac{c(c-1)}{2} = \binom{c}{2}$. The columns of the pairs matrix are indexed by pairs, which is not achievable in MiniZinc, so I created a separate one-dimensional array called $pairs$ whose elements are sets of integers. The code to fill $pairs$ is surprisingly elegant, a simple set comprehension. This array is always filled correctly, that is, the array is in lexicographic order (treating the sets as vectors) from left to right.

```
int: num_pairs = c*(c-1) div 2;
set of int: Pairs = 1..num_pairs;

array[Rows,Columns] of var Colours: grid;
array[Rows,Pairs] of var 1..5: pairs_matrix;
array[Rows,1..4] of var 0..num_pairs: pairs_freq;

% fill the pairs array
array[Pairs] of set of int: pairs =
  [{i,j} | i,j in Columns where i<j];

% channelling constraints
constraint forall(i in Rows, p in Pairs, c in Colours) (
  pairs_matrix[i,p]=c <->
   grid[i,pairs[p][1]] = c /\ grid[i,pairs[p][2]] = c
);

% rectangle constraint
constraint forall(p in Pairs) (
  global_cardinality_low_up([pairs_matrix[i,p] | i in Rows],
    [1,2,3,4], [0,0,0,0], [1,1,1,1])
);

% value precedence
constraint
  let {array[1..r*num_pairs] of var 0..4: x =
      [pairs_matrix[i,p] | i in Rows, p in Pairs]
  } in
  x[1] != 2 /\ x[1] != 3 /\ x[1] != 4 /\
  forall (i in 2..r*num_pairs, c in 2..4)
    (x[i] = c -> exists(j in 1..i) (x[j]=c-1)
);
```

Figure 6.2: MiniZinc code for constraints in the alternative model.

The channelling constraints of Figure 6.2 use $pairs$, along with $pairs\_matrix$ and $grid$. The reader might notice that the channelling constraint in Figure 6.2 is redundant, and can be better expressed as $pairs\_matrix[i,(x,y)] = grid[i,x] \wedge pairs\_matrix[i,(x,y)] = grid[i,y]$. However this form does not take into consideration the value of $pairs\_matrix[i,(x,y)]$. This could be 0, which is not in the domain of $grid$. Modifying the constraint to avoid this involves splitting it into two conditional expressions:

1. $pairs\_matrix[i,(x,y)] = 0 \Rightarrow grid[i,x] \neq grid[i,y]$

2. $pairs\_matrix[i,(x,y)] \neq 0 \Rightarrow pairs\_matrix[i,(x,y)] = grid[i,x] \wedge$
   $pairs\_matrix[i,(x,y)] = grid[i,y]$

This involves explicitly stating that the two grid entries must be different when the associated $pairs\_matrix$ entry is 0, something the channelling constraint in Figure 6.2 does implicitly as the aggregation variable c is taken from the set $Colours = \{1,\ldots,4\}$ and hence never takes the value 0. A few test runs showed that the above version of the constraint is less efficient than the version in the MiniZinc code.

An auxiliary array $pairs\_freq$ is declared to efficiently express *row-set*, whose code is analogous to that of *allperm* in Section 4.4. The Rectangle Constraint is expressed using a weaker version of $global\_cardinality$ called $global\_cardinality\_low\_up$. Instead of fixing the frequencies, this constraint enforces a lower and upper bound on them, given by the third and fourth arguments respectively. Each colour is constrained to appear in each column either 0 or 1 times.

MiniZinc has a global constraint $precedence$ to enforce value precedence. However, the programmer cannot specify the ordering of the values using this constraint. Instead it orders all the values from smallest to largest. The value symmetry in this model is not total, so I wrote my own version specific to this model. It is based on the constraints Law and Lee give for a binary value predence constraint $s \prec t$ over a sequence of variables $<x_0,\ldots,x_n>$ [14]:

1. $x_0 \neq t$ and
2. $x_j = t \Rightarrow \bigvee_{0\leq i<j} x_i = s$ for $1 \leq j < n$

These constraints can be adapted for $1 \prec 2 \prec 3 \prec 4$ over $<x_0,\ldots,x_n>$ as follows:

1. $x_0 \neq 2 \wedge x_0 \neq 3 \wedge x_0 \neq 4$ and
2. $x_j = k \Rightarrow \bigvee_{0\leq i<j} x_i = k-1$ for $1 \leq j < n, 2 \leq k < 4$

A local array of decision variables x, defined using the **let** construct, is used to store the sequence required for value precedence, which in this case is the pairs matrix flattened row-wise. Simply swapping the order of i and p in the array comprehension will flatten it column-wise. The aggregation operator $exists$ is used to express the disjunction in the second constraint.

The first model to be considered is one with all the symmetry breaking constraints applied to the pairs matrix. Balanced solutions will have fewer pairs represented by each colour, so the corresponding pairs matrices will contain more 0s than any other values. Finding the most balanced solutions is equivalent to maximising the number of 0s in the pairs matrix and guiding the solver towards balanced solutions has proven to be successful for the original model. With this in mind, it is very desirable to model this as an optimisation problem, as opposed to a satisfaction problem. However, such a search goal will cause the solver to find all solutions and evaluate them, which involves much more work than terminating search after the first solution is found. Despite this, the search strategy can still be used to guide the solver. Smallest value assignment will prioritise 0 above any other value, so it is a sensible value assignment to use. Both the columns and rows of the pairs matrix are constrained, but since

Size	row-lex+row-prec	row-set+row-prec	no-sym
7×7	43,617 CPs/12.821s	203,334 CPs/55.125s	469,413/58.224s
8×8	15,256 CPs/16.756s	10,169 CPs/12.470s	-
9×9	30,850 CPs/69.318s	41,879 CPs/126.528s	-

Table 6.1: Results of the combined model with smallest value assignment.

the rectangle constraint acts on the columns and propagates quickly, I used a column-wise variable ordering. The results are in Table 6.1.

row-lex+col-prec and *row-set+col-prec* have been omitted from the table as they did not find any 7×7, 8×8 or 9×9 solutions within 10 minutes. In addition, none of the models could find a 10×10 solution in under 10 minutes. One interesting observation to make is that in previous results, the time taken to find a solution increases as the number of choice points increases. However, this is not the case here as the time taken seems to depend more on the grid size than the number of choice points, but I cannot offer any reason as to why this happens.

I also tried the models with random value assignment, as a non-deterministic strategy was effective in previously tested models. A possible disadvantage to this approach is that non-zero values will be assigned roughly 80% of the time, which may slow down search. Models with *col-prec* were not tested due to their poor performance in the previous experiments. The results in Table 6.2 show an improvement, particularly for *no-sym*, which found a 13×13 in under 10 seconds, although it was unable to find a 14×14 solution within 10 minutes.

Size	row-lex+row-prec	row-set+row-prec	no-sym
9×9	10,676 CPs/11.858	724 CPs/1.634s	163 CPs/0.331s
10×10	10,527 CPs/29.756s	-	191 CPs/0.4338s
11×11	-	-	236 CPs/0.602s
12×12	-	-	358 CPs/0.916s
13×13	-	-	2356 CPs/6.900s

Table 6.2: Results of the combined model with random value assignment.

Like the original model, the symmetry breaking constraints were more effective with smallest value assignment, and less effective for random value assignment. Imposing row or column value frequency bounds should give better results as it was effective for the original model. Fixing the value frequencies of the first $\lfloor \frac{N}{2} \rfloor$ rows and the first column of $grid$ was the most effective version of the original model, so similar constraints will be applied to the alternate model. Using the auxiliary array $pairs\_freq$, I implemented these constraints in the same way as I did for the original model. The bounds are easy to compute. If a value appears k times in a row of $grid$, then it will appear $\binom{k}{2}$ times in the corresponding row of the pairs matrix.

Size	no symmetry
10×10	240 CPs/0.482s
11×11	205 CPs/0.637s
12×12	1,722 CPs/4.031s

Table 6.3: Results of the combined model with random value assignment and fixed value frequency bounds for the first $\lfloor \frac{N}{2} \rfloor$ rows and the first column of $grid$.

Table 6.3 shows that the models with symmetry breaking were unable to find any results for 10×10 or larger within 10 minutes, so they are not included in the table.

Random value assignment is better for this model, however the performance worsens as more constraints are added to the pairs matrix. It is particularly difficult to find pairs matrices that satisfy the symmetry breaking constraints. This might be due to the dependencies mentioned earlier, the combination of the row symmetry breaking constraints and value predecence constraints, or both.

However, symmetry breaking has proven to be effective for the original model, so it is advantageous to apply all the symmetry breaking constraints to $grid$, which also removes the option of breaking symmetry on the pairs matrix. This version of the combined model is very close to the original model, the only differences being the inclusion of the pairs matrix and the alternate way of expressing the rectangle constraint, so I don't suspect that searching over $grid$ will speed up search very much, if at all. Searching over the pairs matrix has the most potential gain for this model. However, imposing such tight restrictions on $grid$ will also impose these restrictions on the pairs matrix. With this in mind, it might be detrimental to implicitly constrain the pairs matrix too much, so I tested this model with the following symmetry breaking constraints and row and column value frequencies:

1. *lex2* and fixed value frequencies for the first $\lfloor \frac{N}{2} \rfloor$ rows and the first column of $grid$.
2. No symmetry breaking and fixed value frequencies for the first $\lfloor \frac{N}{2} \rfloor$ rows and the first column of $grid$.
3. *lex2* and no fixed value frequencies.
4. No symmetry breaking and no fixed value frequencies.

The row and column value frequency bounds established in Section 5.2 still apply to any unfixed rows and columns for all of the above models. The constraint *firstrow* is also included in all the models. Column-wise variable ordering and random value assignment was used as this was the best search strategy for the alternate mode. The results in Table 6.4 show that the combined model is very poor when searching over the pairs matrix.

Size	(1)	(2)	(3)	(4)
7×7	10,770,016 CPs/ 381.752s	723 CPs/0.121s	9,876,400 CPs/ 416.333s	161 CPs/0.126s
8×8	-	2,879,408 CPs/84.423s	201,952/6.532s	5,330 CPs/0.333s

Table 6.4: Results of the combined model.

6.4 Adapting the model for 16×16 and 17×17

The original model developed for the general grid-colouring problem was able to find a 15×15 solution in a reasonable time by directing the solver towards balanced solutions. Unfortunately the search space simply becomes too big for any larger grids. Instead of thinking about the general grid-colouring problem, the models can be tailor-made to exploit some properties of 16×16 and 17×17. It is already known that a legal 4-colouring of 17×17, should it exist, has either 4 or 5 of each colour in every row and column [6]. Clearly a similar deduction can be made for 16×16, although the restrictions will be looser since it is a smaller grid.

It has already been proven that a 16×16 grid is 4-colourable, however such a 4-colouring is not presented in the literature. The result is a consequence of the proof that 20×16 is 4-colourable, which was proved using theoretical results as opposed to by example [2]. There are $\binom{16}{2} = 120$ pairs avaliable for each colour in a 16×16 grid. A perfectly balanced 4-colouring will have four of each colour in every row and column. In this case, each colour will occupy $16\binom{4}{2} = 96$ pairs, well within the limit. Now assume one colour appears five times in k rows and four times in every other row. This colour will occupy a total of $k\binom{5}{2} + (16-k)\binom{4}{2} = 96 + 4k$ pairs, so this colour can potentially occur five times in six distinct rows to match the amount of pairs available. By symmetry, these properties are true for the columns as well. This gives a fair amount of freedom, so for the 16×16 model I fixed the domains of *row_freq* and col_freq to be $\{3, 4, 5\}$. It is possible for a 4-colouring of 16×16 to have six or possibly seven of one colour in a row or column, but I did not want to make the bounds too loose as this would increase the size of the search space. I also fixed the row frequencies for the first eight rows. The first four rows have five 1s, four 2s, four 3s and three 4s. The next four rows have four 1s, five 2s, four 3s and three 4s. I ran this model for over four hours but it was unable to find a solution. I ran a similar model where each row and columns were constrained to have exactly four of each colour, but it was also unable to find a solution after several hours.

Despite this, solving 17×17 is not a lost cause. Rectangle-free subsets of 17×17 have been found with maximum cardinality [2] [6]. If a rectangle-free subset were to be assigned a value at compile time, this would reduce the search space considerably. This cannot be achieved in MiniZinc, but the next best option is to apply constraints of the form $grid[x, y] = c$ for every cell (x, y) in the rectangle-free subset. This will reduce the sizes of domains of the associated variables to c at run time, fixing their values. Since such a rectangle-free subset has maximum size, and the previous models have biased towards the smaller values, the cells in the rectangle-free subset will take the value 1. The symmetry breaking constraints also need to be considered. Since *lex2* was the best symmetry breaking model for the larger grids in 5, this

	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17
1	1	1	1	1	1												
2	1					1	1	1									
3	1								1	1	1						
4	1											1	1	1			
5	1														1	1	1
6		1				1			1			1			1		
7		1					1			1			1				
8		1						1			1			1		1	
9			1				1				1				1		
10			1					1		1		1					1
11			1						1				1			1	
12				1		1					1		1				1
13				1			1					1				1	
14				1						1				1	1		
15					1	1				1						1	
16					1		1		1					1			1
17					1			1					1		1		

Figure 6.3: The rectangle-free subset of 17×17 used in the model.

will be the constraint of choice. The rectangle-free subset will have to be consistent with row and column lexicographic ordering. Using Kupin's rectangle-free subset as a starting point (Page 17 of [6]), I applied row and column transformations to obtain the rectangle-free subset in Figure 6.3. Note that permuting the rows and columns is a perfectly legal transformation as it will preserve the rectangle-free property. If the empty cells were filled with 2, say, then the grid would satisfy *lex2*.

74 constraints are required to fix the values of these cells. However, even more can be said about a potential 4-colouring of this format. Both the first row and column will have five 1s followed by four 2s, four 3s and four 4s because of *lex2*. 24 more constraints are needed to fix these values, and these constraints also break the value symmetry. One final addition to the model can be made. The domains of *row_freq* and *col_freq* are $\{4, 5\}$, so there is nothing stopping the solver from assigning a 1 to a variable if its row and column both contain only four 1s. Since all the 1s have been filled in, the solver only needs to assign values from $\{2, 3, 4\}$. While this will probably only cause a small amount of backtracking, I prevented this behaviour with the constraints $row\_freq[i, 1] = 4$ and $col\_freq[j, 1] = 4$ for all rows i and columns j which contain four 1s. The search strategy for this model is as follows: The first row is filled, then the first column. The variables in each are fixed so the value assignment strategy does not matter. Then the 16×16 subgrid obtained by removing the first row and first column is filled row-wise from top to bottom with random value assignment. This model was left to run for over five hours but did not find a solution.

7 Further Work

7.1 Dynamic Symmetry Breaking

Dynamic symmetry breaking methods were not used in this project due to the limitations of MiniZinc. Smith and Gent [11] show that Symmetry Breaking During Search (SBDS) is effective for matrix models, and can be combined with symmetry breaking constraints. This approach will likely be effective for the grid-colouring problem if it were to be modelled in a language which supports SBDS. SBDS uses symmetry breaking constraints, but unlike symmetry breaking constraints, it applies them during search after each assignment is made. This prevents the search space from considering partially assigned grids symmetric to those already visited. Since the symmetry breaking constraints used in the project do not break all the symmetry, I suspect SBDS would improve search greatly by breaking some of the remaining symmetry during search.

7.2 Improving the Combined Model

The combined model was not used to its full potential. For all the tests, the solver searched over the pairs matrix column-wise. It might have been better to search row-wise, or search over $grid$ instead. There may have also been some conflict between value precedence and the row symmetry breaking constraints. Only one set of the row and column value frequency bounds constraints was tried for the combined model, when other sets may have worked better. Implied constraints could have also been implemented to increase propagation speed, such as the two constraints in Section 6.1.

Some properties of the grid-colouring problem were established, both in the literature and in this project, by considering how many pairs would be consumed by a colour if it was assigned to each row or column a fixed number of times. It is known through such reasoning, for example, that it is impossible for a colour to occur more than five times in any row or column of a legal 17×17 4-colouring. It should be possible for this kind of reasoning to be extended and implemented as constraints on the pairs matrix, which will eliminate partial grid assignments in the search space when it becomes apparent that one colour has been assigned too many times, either in one row, or throughout the whole grid, for it to be part of a legal 4-colouring.

7.3 Refining the 17×17 Model

If 17×17 has a 4-colouring, then with some refinement, I believe the model could find one. The current model fixes the 1s in the grid according to a rectangle-free subset and searches over the rest of the grid with the standard search strategy used throughout the project. A potentially better search strategy would be to assign all the 2s in the partially filled grid, then assign all the 3s and all the 4s. To the best of my knowledge, Minizinc does not provide such a value-driven search strategy, but this could be implemented by adding four 17×17 arrays to the model, each representing one of the four colours. The array representing the colour c would contain 1s in the same cells that c occupies in grid. The search strategy could then be changed to search over the arrays to emulate a value-driven search. This model would also have to consider how many times each colour could appear in the grid. Due to time constraints I was not able to implement and test this model.

Furthermore, the rectangle-free colouring of size 74 might not even participate in a 4-colouring of 17×17 if one exists. Removing one of the cells from the rectangle-free subset gives a new rectangle-free subset of size 73. Hence there are 74 such rectangle-free subsets, and each of these could potentially participate in a 4-colouring. A smaller rectangle-free subset is more likely to result in the most balanced 17×17 grid, which is important, especially for grids of this size. Therefore another possibility is to run the model with each of the rectangle-free subsets of size 73 with the strategy described above.

8 Conclusion

Constraint programming has proven effective at modelling and solving the general grid-colouring problem, but not to the extent that was anticipated. A CSP model was written in MiniZinc, and as more features were added, it was able to find 4-colourings for larger grids, peaking at 15×15. Specialised models were designed and implemented for 16×16 and 17×17, utilising the literature on the 4-colouring problem, however solutions were not found in any reasonable time. Constraint programming is a complex discipline. There are many factors to consider, and the way they interact could have a drastic impact on solving the problem. These factors include modelling, constraints, symmetry breaking and the search strategy.

A given CSP could have several possible models, some which perform better than others. Multiple models can be integrated which increases the options available to the programmer. Two models of the grid-colouring problem were considered, one natural and obvious matrix model, and a model which combined the natural model with a second matrix which stored information about the pairs each colour occupied in each row. Without any symmetry breaking constraints, the combined model was slightly better, but the symmetry breaking constraints and value frequency bounds worked far better for the natural model. As mentioned in the previous section, there are many more avenues to explore for the combined model.

There are often many different ways to express a constraint, especially if multiple models are considered, some more efficient than others. Four different ways of writing the rectangle constraint of the original model were compared in Chapter 3, and the best constraint, $corner$, was used for the rest of the project. An additional way of expressing the rectangle constraint was introduced with the alternate model. Symmetry breaking constraints were also used to break symmetries of the problem, the most significant of these being row and column symmetries. These symmetries have been thoroughly studied and many constraints have been developed to break them. A few of these constraints were tested in conjunction with constraints which broke rotation/reflection symmetries and value symmetry. The search space for this problem is very solution-rich, so the symmetry breaking constraints didn't significantly improve search until row and column value frequency bounds were introduced to reduce the search space further. Models that incorporated symmetry breaking were generally less effective than the models without symmetry breaking. However, some of the symmetry breaking models found solutions for 15×15 quite quickly, something that the models without symmetry breaking were unable to accomplish. This makes sense as the number of symmetries increases super-exponentially with the size of the grid, so the symmetry breaking models had a much greater effect as the grid size increased. One interesting result was that the model using *lex2* more often than not found a 15×15 solution faster and with less backtracks than the stronger model which used $lex2 + allperm$. This suggests that for the grid-colouring problem, and perhaps for other problems with a vast amount of solutions, it is easy to over-constrain the problem instead of constraining it just enough to find the first solution in the most efficient

manner, something that I have never come across before in constraint programming. A delicate balance is required.

The search strategy is one of the most important factors. Both variable ordering and value assignment have to agree with the model and constraints. This was seen in Chapter 3, where variable ordering had a large impact on the performance of the many corner comparison constraints in Section 3.2. Despite this, not many search strategies were used in this project, and the ones used were fairly basic. Only row-wise and column-wise variable orderings were considered, and the value assignments used were limited to smallest value assignment and random value assignment. In hindsight, experimenting with the other solvers that MiniZinc supports and experimenting with more complicated search strategies would have likely improved the model.

MiniZinc was the language of choice for this project. While not as established as other constraint modelling languages such as Eclipse or the Gecode and ILOG C++ libraries, it is an easy-to-learn and intuitive language which allowed me to capture the problem and its constraints in a succinct manner. While it does offer some freedom in features such as user-defined constraints and search annotations, it does have several limitations. The main limitation is its lack of search strategies, at least for the default solver. MiniZinc doesn't offer a large number of variable orderings and value assignments to choose from. However, MiniZinc is supported by a number of external constraint solvers that might support more search strategies, but I only used the default solver which limited my options further. Another limitation of MiniZinc is that it doesn't support any dynamic search features, making it impossible to implement any dynamic symmetry breaking methods which may have proved effective, especially in conjunction with symmetry breaking constraints.

Bibliography

[1] William Gasarch, *The 17×17 challenge*, November 2009. [Online] Available at: http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html [Accessed November 26, 2010.]

[2] Stephen Fenner, William Gasarch, Charles Glover and Sammy Purewal, *Rectangle Free Coloring of Grids*, [Online] Available at: http://www.cs.umd.edu/~gasarch/papers/grid.pdf [Accessed November 26, 2010.]

[3] Willem-Jan van Hoeve and Irit Katriel, "Global Constraints", in *The Handbook of Constraint Programming*, 1st ed., Elsevier Science, 2006, pages 169-208.

[4] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, Guido Tack, "MiniZinc: Towards a Standard CP Modelling Language", in *Principles and Practice of Constraint Programming - CP 2007* Heidelberg: Springer-Verlag 2007, pages 529-543.

[5] Peter J. Stuckey et al., "The Evolving World of Minizinc", in *Constraint Modelling and Reformulation (ModRef 09)*, September 2007, pages 156-170.

[6] Elizabeth Kupin, *Notes on 4-coloring the 17 by 17 grid*, August 2009. [Online] Available at: http://www.cs.umd.edu/~gasarch/BLOGPAPERS/bethk.pdf [Accessed November 26, 2010.]

[7] Håkan Kjellerstrand, *Nontransitive dice, Ormat game, 17x17 challenge*, August 2010. [Online] Available at: http://www.hakank.org/constraint_programming_blog/2010/08/nontransitive_dice_ormat_game_17x17_challenge_1.html [Accessed December 13, 2010.]

[8] Ian P. Gent, Karen E. Petrie, Ralph Becket and Jean-Franǫis Puget, "Symmetry in Constraint Programming", in *The Handbook of Constraint Programming*, 1st ed., Elsevier Science, 2006, pages 329-376.

[9] Pierre Flener et al., *Breaking Row and Column Symmetries in Matrix Models*, 2002.

[10] Y.C. Law and J.H.M.Lee "Symmetry Breaking Constraints for Value Symmetries in Constraint Satisfaction", in *Constraints* Netherlands: Springer 2006, pages 221-267.

[11] Barbara Smith and Ian P. Gent, *Reducing Symmetry in Matrix Models: SBDS v. Constraints*, 2001.

[12] Alan M. Frisch, Christopher Jefferson and Ian Miguel, “Constraints for Breaking More Row and Column Symmetries”, in *Principles and Practice of Constraint Programming - CP 2003*, Heidelberg: Springer-Verlag 2003, pages 318-332.

[13] Zeynep Kiziltan and Barbara M. Smith, “Symmetry-breaking Constraints for Matrix Models”, in *Proceedings of the Second International Workshop on Symmetry in Constraint Satisfaction Problems (SymCon02)*, September 2002.

[14] Yat Chiu Law and Jimmy H.M. Lee, “Global Constraints for Integer and Set Value Precedence”, in *Principles and Practice of Constraint Programming - CP 2004*, Heidelberg: Springer-Verlag 2004, pages 362-376.

[15] Barbara M. Smith, “Modelling”, in *The Handbook of Constraint Programming*, 1st ed., Elsevier Science, 2006, pages 377-406.