CS4402 Constraint Programming
Practical 1: Modelling

Student Id: 150025279
University of St Andrews

March 6, 2017

1 Introduction

This report summarises the work done for the Practical Assignment #1 of the CS4402 Constraint Programming module, whit the goal to create and evaluate a model for the Rook Placement Puzzle. The problem that is being solved is described, as well as a detailed description of the design and implementation of the solution in the Essence Prime language. Also, an evaluation of the solution against a big instance of the puzzle using the different optimisation levels and heuristics available in Savile Row is presented as part of this report.

2 Problem Statement

The main problem of the practical is to write a constraint model in Essence Prime for the Rook Placement Puzzle, which is described as follows: Given an nxn board with three types of squares on it (empty squares, blocking squares and cue squares), place a set of rooks on the board such that:

1. No rook is placed on a blocking square or clue square
2. Every clue square containing the number i has exactly i rooks in adjacent squares. The adjacent squares are those above, blow, left or right of the clue square.
3. No pair of rooks attack each other
4. Every empty square on the board is attacked by some rook.

As in chess, a rook can move any distance along a row or column, and additionally, blocking and clue squares block the attacks of the rooks in this puzzle.
Some instances of the Rook Placement Puzzle present vertical and/or horizontal flip symmetry. The model should be extended to detect when the board has any type of symmetry and break it (only when it is present in a given instance of the problem).
Finally, an empirical evaluation of the performance of the model on a range of instances should be presented, using the different levels of optimisation and heuristics provided by Savile Row. Additionally, a comparison between the model's performance with and without symmetry breaking needs to be performed.

3 Model

This section describes the Essence Prime model that was created for the Rook Placement Puzzle, including the source code structure, parameters, decision variables and constraints of the model, as well as the approach taken to detect and break the symmetry of the problem.

3.1 Source Code

The source code and the parameters for the problem instances presented in this report are located in the *code* folder found in the submission for this practical. This folder contains the following files and sub-folders:

- *rooks.eprime*: Essence Prime model for the Rook Placement Puzzle (without symmetry breaking)
- *rooks-sym.eprime*: Extension of the model in rooks.eprime to include the detection and breaking of symmetry.
- *rooks-sym-h.eprime*: Extension of the model in rooks-sym.eprime to include the use of heuristics.
- *params/*: Contains the .param files that represent the problems described and evaluated in this report.

3.2 Parameters

The implemented model follows the standard format of the parameter file specified for this assignment, without adding new additional parameters:

- n: the size of the board
- $squareType$: an nxn matrix to indicate the type of each cell in the board: 0 represents a blank square, 1 a blocking square, and 2 a clue square. This matrix will be represented as T in this report for simplicity.
- $clues$: an nxn matrix that represents the amount of rooks that need to be around every clue cell. This matrix will be represented as C in this report for simplicity.

The model validates the relation between the $squareType$ and $clues$ matrix with the following boolean expression using the *where* keyword in Essence Prime:

$$\forall i \in [1,n] \forall j \in [1,n] : (T_{ij} \leq 1 \implies C_{ij} = 0)$$

Meaning that, if a square is empty or a blocking square, the corresponding square in $clues$ should be 0 (it is not necessary to validate that the value for cue squares is between 0 and n since this is specified in the domain of the parameter). This is important to validate in order to correctly determine if an instance of the problem is symmetric or not (as it will be explained in a later section)

3.3 Decision variables

The solution of the Rook Placement Puzzle is given by a single decision variable called *rooks* (identified as R in this report), where R is an nxn matrix with an integer domain from 0 to 1. If $R_{ij} = 1$ it means that there is a rook in the row i and column j of the board, and $R_{ij} = 0$ means that a rook is not placed in that square.

3.4 Constraints

Constraint #1: No rooks are placed on blocking or cue squares

This constraint can be interpreted as if the type of a square is different than the empty square, then that cell should have a value of zero in the solution matrix (no rook):

$$\forall i \in [1,n] \forall j \in [1,n] : T_{ij} \neq 0 \implies R_{ij} = 0$$

Constraint #2: Clue squares have exactly i rooks in adjacent squares

This constraint can be written as that, for a square in row i and column i, the sum of the values of the adjacent squares in the solution matrix *rooks* should be equal to the value in C_{ij}:

$$\forall i \in [1,n] \forall j \in [1,n] : T_{ij} = 2 \implies R_{i,j+1} + R_{i,j-1} + R_{i-1,j} + R_{i+1,j} = C_{ij}$$

Note that, when an index is out of bounds for an integer matrix in Essence Prime, the value of the expression will be undefined. This is handled in the model by summing the results of the expressions $rooks[i, j+1] = 1$ instead of the value of rooks, given that Essence Prime treats a boolean value of true to be one, and false to be zero, and that the expression $undefined = 1$ will evaluate to false, producing the sum of the values adjacent to a square even when it is at the border of the board.

3.4.1 Constraints #3 and #4: Rooks attacks

In order to comply with the constraints stating that no pair of rooks attack each other and that every empty cell must be attacked by some rook a similar approach is taken and they are described together.

Suppose that it is needed to check if the square R_{ij} (independently if there is a rook in that square or not). The high level idea of the constraint can be described as follows:

1. Determine the *attack zone* in the row i and in the column j of the square R_{ij}. That is, the squares which a rook could attack if $R_{ij} = 1$, or the squares from where a rook could attack R_{ij} if $R_{ij} = 0$
2. If $R_{ij} = 1$, then the amount of rooks placed in the attack zone **must** be equals to one (both in row i and column i), guaranteeing that no pair of rooks attack each other (note that R_{ij} is taken into consideration in the *attack zone*). Otherwise, if $R_{ij} = 0$, then the total number of rooks in the *attack zone* must be greater than one, guaranteeing that R_{ij} is attacked by at least one rook.

A square s is in the *attack zone* of R_{ij} if: it is in the same row i or in the same column j, and if all the squares between s and R_{ij} are empty. In order to determine this, the definition of the matrix squareType T is used: if the sum of the values in T between the square s and T_{ij} is **equals** to zero, then it is in the attack zone (all the squares are empty and thus, the sum of the types must be equals to zero).

For example, assume that it is needed to the determine the attack zone of R_{ij}. The constraint will analyse the row first and "create" the vector A where $A_k = true$ if R_{ik} is in the attack zone (and false otherwise) as follows for the left side of the row:

$$\forall e \in [1, j] : A_e = ((\sum_{c=e}^{j} T_{ic}) = 0)$$

And for the right side of the row:

$$\forall e \in [j, n] : A_e = ((\sum_{c=j}^{e} T_{ic}) = 0)$$

The Vector A provides boolean values for whether the squares in row i are within the *attack zone* of R_{ij}. Similarly, a vector B can be defined for the attack zone in column j as follows ($B_k = true$ if R_{ki} is in the attack zone and false otherwise):

$$\forall e \in [1, i] : A_e = ((\sum_{c=e}^{i} T_{cj}) = 0)$$

And:

$$\forall e \in [i, n] : A_e = ((\sum_{c=i}^{e} T_{cj}) = 0)$$

A and B can then be used to determine the amount of rooks that are present in the attack range of R_{ij}. Constraint #3: "No pair of rooks attack each other" can be expressed with the following logical expression:

$$\forall i \in [1, n] \forall j \in [1, n] : T_{ij} = 0 \wedge R_{ij} = 1 \implies \sum_{c=1}^{n} (A_c * R_{ic}) = 1 \wedge \sum_{r=1}^{n} (B_r * R_{rj}) = 1$$

Meaning that there can only be one rook in the row attack zone and one rook in the column attack zone (which is R_{ij}). Similarly, Constraint #4: "Every empty square on the board is attacked by some rock" can be expressed using A and B as follows:

$$\forall i \in [1, n] \forall j \in [1, n] : T_{ij} = 0 \wedge R_{ij} = 0 \implies \sum_{c=1}^{n} (A_c * R_{ic}) > 0 \vee \sum_{r=1}^{n} (B_r * R_{rj}) > 0$$

Meaning that there should be more than one rook in the attack zone of the current square.

The implementation of the constraints in Essence Prime can be found in the source code submitted with this report, each one of them identified with comments as Constraint #1, Constraint #2, Constraint #3 and Constraint #4.

3.5 Symmetry Breaking

As stated in the problem description, an instance of the Rook Placement Puzzle can have horizontal symmetry, vertical symmetry, both of them or no symmetry at all. The model presented in sections 3.2 to 3.4 has been extended in the file *rooks-sym.eprime* in order to break horizontal flip and vertical flip symmetry when they are present in an instance of the problem. In order to do so, the following decision variables were added to the model:

- *horizontalSym*: Decision variable with a boolean domain that indicates if horizontal flip symmetry is present in the model or not.
- *verticalSym*: Decision variable with a boolean domain that indicates if vertical flip symmetry is present in the model or not.

Horizontal flip symmetry is present on a board if, when divided in half, both sides of the board "reflect" each other, i.e., column 1 is equals to column n, column 2 is equals to column $n-1$, ..., and column $\lfloor n/2 \rfloor$ is equals to $n - \lfloor n/2 \rfloor + 1$. Note that, for boards where n is odd, the column $\lfloor n/2 \rfloor + 1$ is not evaluated as it is the exact center of the board and not important to determine symmetry. Also, in order to determine if two columns are the same, it is necessary to take in consideration the types of the squares as well as the value i of each of the clue squares. Similarly, the vertical flip symmetry can be detected using the same approach but comparing the equality of the rows instead of the columns.
With that, the presence of horizontal symmetry can be defined with the following expression:

$$horizontalSym = (\sum_{i=1}^{n} \sum_{j=1}^{\lfloor n/2 \rfloor} |(T_{ij} + C_{ij}) - (T_{i,n-j+1} + C_{i,n-j+1})|) = 0$$

And following the same structure, the presence vertical symmetry can be defined with:

$$verticalSym = (\sum_{i=1}^{\lfloor n/2 \rfloor} \sum_{j=1}^{n} |(T_{ij} + C_{ij}) - (T_{n-i+1,j} + C_{n-i+1,j})|) = 0$$

Once the presence of symmetry has been determined, the *lex* operator can be used to force the left (or top) half of the board to be lexicographically less than the right (or bottom) half of the board. This will cause the removal of solutions where the halves of the board are inverted. To express this constraint in Essence Prime, two one dimensional matrix are created with the elements of the halves of the square (in a reflective order). Let A be a one dimensional matrix with the elements of columns 1 to $\lfloor n/2 \rfloor$ of the solution matrix R, and B the one dimensional matrix with the elements of columns n down to $n - \lfloor n/2 \rfloor + 1$ (in that order) of R, horizontal flip symmetry can be broken with the following expression:

$$horizontalSym \implies A \leq_{lex} B$$

To break the vertical flip symmetry, A is defined as the one dimensional matrix containing the elements of the rows 1 to $\lfloor n/2 \rfloor$ of the solution matrix R, and B the one dimensional matrix with the elements of the rows from n down to $n - \lfloor n/2 \rfloor + 1$, and the constraint is written as:

$$verticalSym \implies A \leq_{lex} B$$

4 Evaluation

In order to evaluate the model that was created and the effectiveness of the constraints, a set of 6 different $5x5$ boards where created in order to examine different edge cases for the Rook Placement Puzzle, all of them illustrated in Figure 1. The parameter files for this puzzles can be found in the *code/params/* folder submitted along this report.

The boards were created so that they tested different aspects of the constraints:

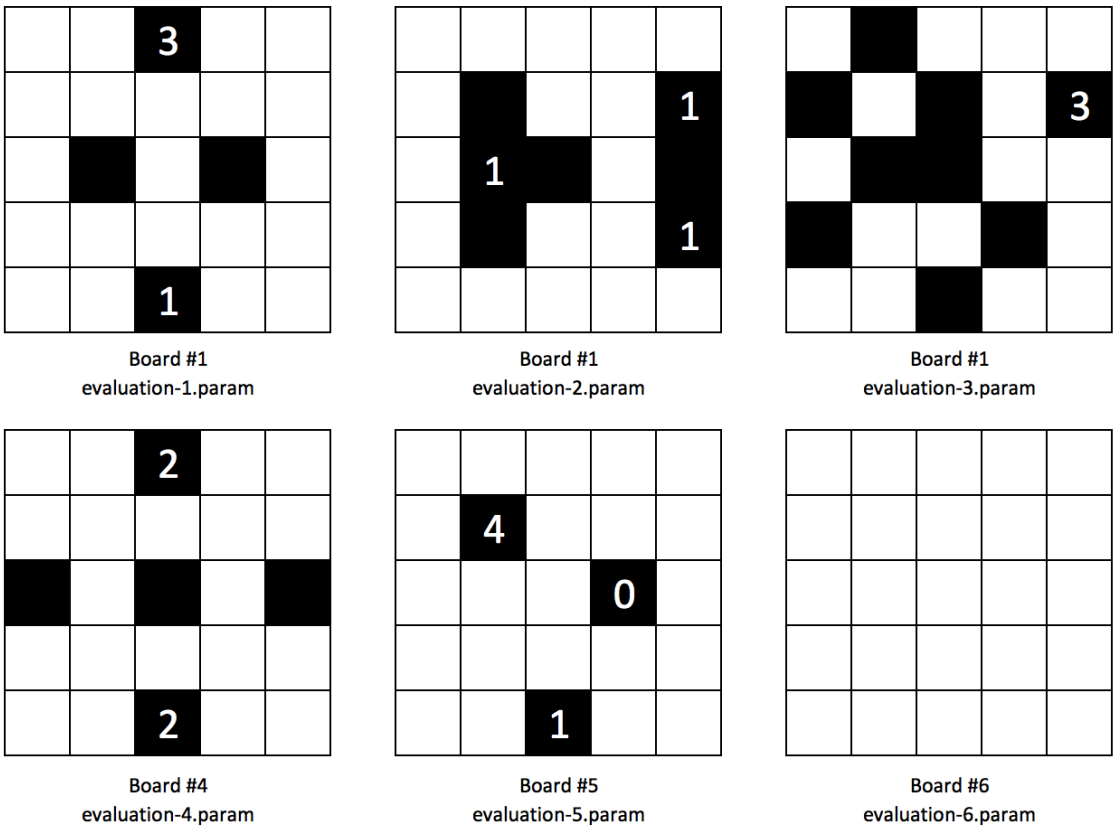

Figure 1: 5x5 boards configuration used for evaluation.

- Different types of squares: the boards contain all the different types of squares (empty, blocking and cue), and all possible i values for the cue squares (from 0 to 4).
- Different locations for blocking and cue squares: blocking and cue squares are placed on the edges of the boards as well as in the middle of them. They're also placed next to each other.
- Different "attack zones": Different sizes and orientations for the "attack zones" as described in section 3.4.1: sizes from 1 to n, vertically and horizontally, as well as having clue and blocking squares delimiting these zones.

4.1 Basic Model Evaluation

The instances described are first tested with the model in *rooks.prime* (without symmetry breaking constraints). Table 1 displays the results and the statistics for these runs: the number of solutions found by the solver, the time taken by Savile Row, the solver solve time, the solver total time and the number of nodes used by the solver in order to find the solution.

Board	Solutions	Solver Total Time	SR Total Time	Solver Solve Time	Nodes
#1	6	0.002019	0.958	0.000196	16
#2	5	0.002592	0.989	0.000153	9
#3	3	0.000565	0.997	0.000022	5
#4	2	0.001928	0.956	0.000059	3
#5	1	0.000317	0.967	0.000008	1
#6	120	0.015505	1.025	0.008416	571

Table 1: Results for the evaluation of the Rook Placement Puzzle.

In all cases, the solver was able to find all the solutions to the problems, from 1 for board #5 to 120 for board #6 (a board containing only empty squares). In order to generate and evaluate the solutions, place the source code folder in the same directory as Savile Row and run the following command, replacing i with the number of the board to evaluate:

./savilerow code/rooks.eprime code/params/evaluation-i.param -run-solver -all-solutions

4.2 Symmetry Breaking Evaluation

The 6 instances of the problem are evaluated using the *rooks-sym.eprime* model as well, in order to evaluate the ability of the model to detect and break both horizontal and vertical symmetry when it is found in the boards. Notice that board #1 presents horizontal flip symmetry, board #2 presents vertical flip symmetry, boards #4 and #6 have both types of symmetry and boards #3 and #5 do not present symmetry.

Board	H. Sym	V. Sym	Solutions	Solver Total T.	SR Time	Solver Solve T.	Nodes
#1	TRUE	FALSE	3	0.001236	0.948	0.000036	5
#2	FALSE	TRUE	3	0.001342	0.936	0.00005	5
#3	FALSE	FALSE	3	0.00092	1.626	0.000036	5
#4	TRUE	TRUE	1	0.000254	0.914	0.000007	1
#5	FALSE	FALSE	1	0.000323	0.969	0.000006	1
#6	TRUE	TRUE	50	0.010364	1.111	0.003201	219

Table 2: Results for the evaluation of the symmetry breaking constraints.

Table 2 shows the results and the statistics for the boards configuration taking symmetry into consideration. Notice that the decision variables that indicate if there is horizontal or vertical flip symmetry (H. Sym and V. Sym columns in Table 2) are assigned correctly by the solver. A decrease on the Solver Total Time and Solver Solve Time is present compared to the times it took the model without symmetry breaking to solve the problems (Table 1), but the Savile Row total time remains very similar for both cases (except for board #3, which takes longer when symmetry breaking is present).

The number of solutions found when there is symmetry is reduced by half or more in all cases. It is interesting to note that for board #3, the total number of solutions is 5, and with symmetry breaking enabled it gets down to 3. This happens because there is one solution where the top part of the board is populated by rooks in exactly the same way as the bottom part of the board. It is important to note that this solution is not lost after the symmetry constraints are introduced.

In order to generate and evaluate the solutions, place the source code folder in the same directory as Savile Row and run the following command, replacing i with the number of the board to evaluate:

./savilerow code/rooks-sym.eprime code/params/evaluation-i.param -run-solver -all-solutions

4.3 Savile Row Optimisation Evaluation

In order to evaluate the optimisation options offered by Savile Row and the impact they have in the amount of time and number of nodes that it takes to find the solutions to a problem, a new and more complex board was created, where the impact of these options can be significant. Figure 2 shows the configuration of the board used for this evaluation.

The board was tested using all four different optimisation levels and the model that takes into consideration the symmetry of the board. Table 3 shows the results of the evaluation. To generate and evaluate the solutions, place the source code folder in the same directory as Savile Row and run the following command, replacing i with the number of the optimisation option to be used (from 0 to 3):

./savilerow code/rooks-sym.eprime code/params/evaluation-7.param -run-solver -all-solutions -Oi

Optimisation	Solutions	Solver Total Time	SR Total Time	Solver Solve Time	Nodes
O0	74314	5.16284	0.849	5.10212	166543
O1	74314	4.92903	1.011	4.87433	166543
O2	74314	3.52452	1.708	3.48587	166543
O3	74314	3.44463	1.802	3.42653	166543

Table 3: Results for the different optimisation levels present in Savile Row.

The solution shows that all the optimisation levels produces the same number of solutions using the same number of nodes, but O0 presents the worst total time, while O3 produces the best one.

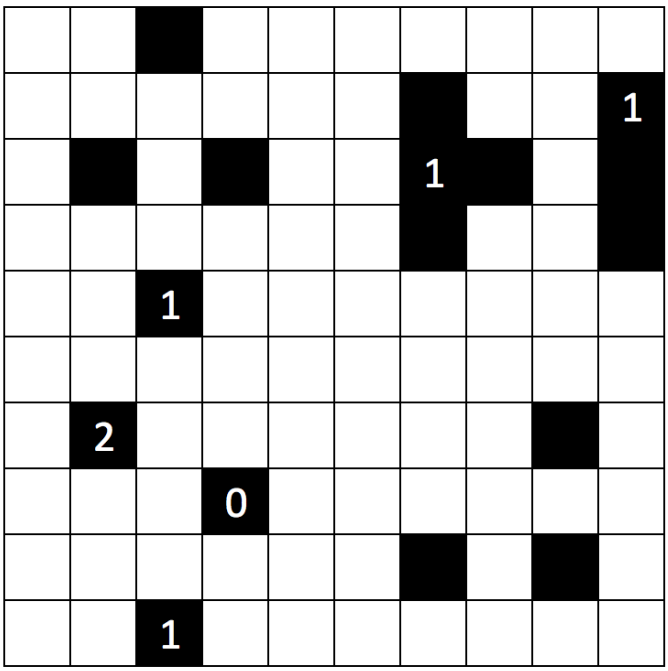

Figure 2: 10x10 board configuration used for evaluation of optimisation levels and heuristics

The total time of O2 is close to the one of O3. It is interesting to note that while O3 and O2 have the best solver total time, the Savile Row Total Time is significantly greater than those produced by O0 and O1, but the problem takes less time to solve.

4.4 Savile Row Heuristics Evaluation

In order to evaluate the heuristics available in Savile Row, the board exemplified in Figure 2 is used, as well as the level of optimisation that produced the best results (O3).

Heuristic	Solutions	Solver Total Time	SR Total Time	Solver Solve Time	Nodes
None	74314	3.44463	1.802	3.42653	166543
sdf	74314	3.10186	1.883	3.08106	166543
static	74314	2.89113	1.652	2.87257	166543
conflict	74314	3.06339	1.895	3.03947	166543
srf	74314	3.05753	1.837	3.03723	166543

Table 4: Results using the different heuristics options.

Table 4 presents the results of using the different available heuristics. All the heuristics were tested using the line *"branching on [rooks]"*. To generate and evaluate the solutions, place the source code folder in the same directory as Savile Row and run the following command (the source code is using the static heuristic):

./savilerow code/rooks-sym-h.eprime code/params/evaluation-7.param -run-solver -all-solutions -O3

The results in Table 4 show an improvement in the total time needed to solve the problem with all the heuristics, compared with the results in Table 3 where none was used. Among the heuristics, *static* is the one that decreases the total time the most, with a reduction in both the Savile Row total time and Solver Solve Time. Note that the number of solutions found and the amount of nodes that it takes to find them does not change while using different heuristics.

4.5 Conclusion

A model for the Rook Placement Puzzle was created and evaluated, showing that it successfully finds solutions for a range of instances of the problem. Additionally, the impact of the different optimisation levels and heuristics present in Savile Row was evaluated when applied to a large instance ($n = 10$) of the Rook Placement Puzzle and the model that was created. Under this circumstances, an optimisation level of O3 and the *static* heuristic produce the best result. It can not be said that this configuration will produce the best results for all the instances of the problem, since they are treated as a black box and their impact might change on problem instances with different characteristics (board size, amount of blocking and cue squares, etc.)