CS4402
Constraint Programming
Practical 1: Modelling

ID: 090017799

March 2017

1 Introduction

The main task of the practical is to implement the constraint model for Rook Placement Puzzle that consists the square board and only one acting figure - i.e. chess rook. The board and the rook have several conditions which need to be considered in the problem as constraints that need to be satisfied without breaking each other. The program should introduce the variables and parameters to use them to write the constraints using the Essence Prime language.

2 Aim

The aim of the practical is to understand the concept of solving the constraint satisfaction problem (CSP) about the chess rook on the n*n board. The constraint modelling should be done using the Essence language, which was implemented by the tool Savile Row. To learn the process of evaluation of model performance using the Savile Row tools and its enhancement. As well as to deeply understand the concept of the symmetry breaking again using the provided tools of the Savile Row.

3 Design

The design of the model strictly follows the conditions of the practical specification. Where every condition is considered as a constraint of the model. Based on these constraints the model should produce all possible correct solutions to the specification.

3.1 Part 1: A Model of the Rook Placement Puzzle

The practical was consisting of the n*n board with three types of squares,

- '0' - empty square (coloured white)
- '1' - blocking square (coloured black)
- '2' - clue squares (coloured black with integer value that ranges between 1..4)

Each of these squares has their own domains, in order not to confuse between their representation and their value from the domain. It was needed to create several square n*n integer 2-dimensional matrices. Thus two matrices were already given in the practical specification:

- "squareType" - for storing three types of squares in the integer domain {0..2}
- "clues" - for storing the values of each clue in the appropriate position of the board, and all non-clues are denotes with zeros, this the integer domain here is {0..4}
- "solutionBoard" - for storing the rooks only, the domain could be {0..1}, where the rooks are denoted with '1' and all others with '0'.

The conditions of placing these types of squares and their values in other corresponding matrices are considered as constraints of the models, as well as the general conditions of the puzzle:

- Every empty square in the "squareType" matrix could have rook;
- Every clue should be surrounded with that number of rooks as the number specified in it;
- No pair of rooks attack each other (i.e. they cannot be placed in the same row or column if they are not separated by some block or clue horizontally or vertically);
- Every empty square should be attacked by any of the rook at least once.

4 Implementation

The implementation of model is done in the Essence language.

4.1 Part 1: A Model of the Rook Placement Puzzle

There is a file with the extension **.eprime** called **"rookPlacementPuzzle.eprime"**, which essentially contains the implementation for CSP of the rook puzzle in the Essence language.

At the beginning of the program it was needed to declare the parameters for the size of the matrices and the matrices with their domains for the boards. The initialisation of these parameters have been done in the file with the extension **.param**. In the folder with the implementation files several examples could be found.

All possible solutions for this model should be saved in some matrix, which will demonstrate the location of all possible rooks satisfying the conditions of specification.

After the keywords "**such that**" the constraints of the model have been specified. In overall there were implemented five constraints:

1. Every empty square of the **"squareType"** matrix might have the rook (which is '1' in the textbf"solutionBoard" matrix);

2. If the **"squareType"** matrix has a clue, then its vales in the **"clues"** matrix should be equal to the sum of rooks in the adjacent squares (i.e. rooks == '1' from the **"solutionBoard"**). This constraint has several cases depending on the location of the clues on the board (i.e. in the middle of the board, in the edge or in the corner).

3. This constraint checks that every square in the **"squareType"** matrix has no rooks on the adjacent squares in the **"solutionBoard"**. It also has several cases as the previous constraint, depending on the location of the rook (i.e. in the middle of the board, in the edge or in the corner).

4. This constraint checks that every empty square from the **"squareType"** has been attacked at least once by some rook in the **"solutionBoard"**. Therefore if the row or column already has some rook it may have more rooks, if they are separated by block or clue and not attacked from the other rook in the column.

5. This constraint checks that every empty row and column may have some rook on it.

4.2 Part 2: Empirical Evaluation

This part is implemented using the Savile Row tool to run all types of heuristics which can be accessed (see the following table). In this example the given example form the specification was used.

Size of board (n)	Heuristic type	Nodes	Time
n=5	static	16	0.000742
n=5	sdf	16	0.000773
n=5	conflict	16	0.000729
n=5	srf	16	0.000775

The table above shows that the heuristic optimisation does not affect the number of nodes and does slightly affect the time of the implementation.

Size of board (n)	Optimisation type	Nodes	Time
n=5	-O0	16	0.002
n=5	-O1	16	0.001317
n=5	-O2	16	0.001162
n=5	-O3	16	0.000756

Accordingly the table above, which was experimented on the given example of the practical specification with better level of optimisation the total time of solving is improving (i.e. it gets faster).

4.3 Part 3: Symmetry Breaking

In order to find the potential improvement of the model there is a concept to break the symmetry of the model. The specification of the practical requires to implement only two types of the symmetry breaking: horizontal flip break and vertical flip break. This means that the **"solutionBoard"** is flipped vertically and horizontally. Thus if it was flipped vertically, everything what is in the first column of the original matrix should appear on the last column of the new matrix; and what is in the second column of the original matrix should appear in the **size-1** column of the new matrix. If it was flipped horizontally, then the same process happens in terms of the rows.

In order to implement the constraints for the vertical and horizontal symmetry breaking, there were two more matrices were declared: **vertSymmetryBoard** and **hortSymmetryBoard** with the same size as the original one, in this case the **solutionBoard** matrix.

The indexes of the row should stay the same, while the indexes of the column in the should be determined by the following formula: **((n+1)-col)**, where the **n** is the size of the matrix, and **col** is current index of the column. The same idea is used for the horizontal symmetry, where the column stays the same and the row is changes as **((n+1)-row)**.

Size of board (n)	Type of Experiment	nodes	Time
n=5	without symmetry break	16	0.000765
n=5	horizontal symmetry break	3	0.000441
n=5	vertical symmetry break	5	0.000728
n=5	hor-l and vert. sym. breaks	3	0.00047

Accordingly the table above it can be concluded that the symmetry breaking improves the performance of the model. It can be seen that horizontal symmetry improves the model better than the vertical one. Since the number of nodes with the horizontal one decreased from the 16 to 3 nodes, while with the vertical one from 16 to 5 nodes. In terms of the time performance, with less nodes the horizontal symmetry breaking works faster. If both these symmetries will be run together it will expands only up to 3 nodes while the time is slightly slower than with the horizontal symmetry break only.

5 Testing

The practical was tested in several examples of boards, where the blocks and clues are located differently and with different numbers for the clues. All of the values for the parameters must be valid. For instance, the clue values should not exceed the number of possibles sides. All produced solutions are correct, which means that all possible constraints are considered.

Moreover the model still works for the different sizes as well.

6 Evaluation

During the implementation of the model there were some limitation with considering different examples. Where the model could solve the given example in the specification, but wasn't producing the solutions for any other examples. It was concluded that the model could have too much constraints, which are overconstraining the model, and it cannot produce the solution at all.

Sometimes it also could be that not all possible constraints are considered and the model produces too many solutions, some of which are not correct, i.e. having not satisfied conditions.

7 Conclusion

The overall practical very useful for deep understanding of the constraint programming concepts as well gaining the practice in modelling of the CSPs in a new language Essence.

Using this language the code was always improved to generalise to all possible other examples, which satisfies the conditions and produces only correct solutions.