

# CS4402 - Practical 1

## The Rook Placement Puzzle Constraint Modelling

150025289

March 2017

### 1 Introduction

This project is about writing a constraint model of a given problem on the language `Essence Prime` and do experiments on it using `Minion Solver` and `SavileRow` which captures the language interpretation and passes to Minion.

### 2 Problem

The given problem for this practical is called *The Rook Placement Puzzle*. It has an chess-like board with flexible size of  $n \times n$  and it consists the placements of rooks depending on the given *blocking* and *clue* squares. The blocking squares block the attack conditions of the rooks. The clue rooks also act like blocking squares but they also require the specified number of rooks in their 1 block neighbourhood. Additionally, the rooks exactly like they act in chess and threaten every square in the same row or column by the puzzle definition.

### 3 Model

In the definition of the problem, the input parameters are defined as 2  $n \times n$  matrices which specify blocking/clue squares and if they are clue squares the required number for that clue. In this point, we made an assumption that the parameters are given correctly which means, if a square is indicated as a clue square, the second matrix must have a non-zero value associated to it.

For the first matrix `squareType`, the domain is given as the integers from 0 to 2. 0 means an empty square, 1 indicates a blocking square and 2 represents a clue square. The second matrix `clues` is limited to the integers from 0 to 4 which represents the number of the rook neighbours wanted.

Using the same principle, we can define our output value matrix as a  $n \times n$  with boolean like integer values (0, 1). This means that we are choosing the use occurrence representation. We believed that it would be more logical to choose like this since the input values are given like this and it would be easier to make the connection between them.

## 4 Constraints

As the puzzle defines, we have a couple of rules to satisfy. We can split up this rules as constraints and discuss them individually.

### 4.1 No Rooks Attack each Other

The puzzle requires that any rook can't threat another one. To make this happen, we can go through all the rooks and check their relations with other rooks. This would be the case if we used an explicit representation for the rooks placements and indicate every rook placement with one element. Since we are using the occurrence representation for the rooks and represent the rooks by their existence in the  $2D$  matrix, we need to take another approach.

We can go through all the elements in the the board twice and compare them if they have rooks in it. First of all we need to make sure that this two elements are in the same row or column but not the same row and column at the same. Logically, we can represent like in the equation 1.  $r_1$  and  $r_2$  represent the 2 rows and  $c_1$  and  $c_2$  represent the 2 columns. This logical expression is actually an **XOR** and ensures that one of the equalities is satisfied but not both of them at the same time.

$$((c_1 = c_2) \vee (r_1 = r_2)) \wedge ((c_1 \neq c_2) \vee (r_1 \neq r_2)) \quad (1)$$

After using this as a base check, we can check the rooks matrix if the `rooks[r1, c1]` and `rooks[r2, c2]` have 1 values. If this check is also satisfied we need to check the in between squares to see if there is a blocking/clue square in the middle. We should forbid any case which 2 rooks can see each other without any blocking/clue squares in the middle of them.

We can make this happen by using a max function depending on the row/column case. If they are in the same column for instance ( $c_1 = c_2$ ), we can go from  $r_1$  to  $r_2$  and search a max value for square type and force it be more than 0. This logic can be seen in the equation 2.

$$\begin{aligned} ((c_1 = c_2) \implies \max(\text{squareType}[i, c_1] \mid i \in \text{int}(r_1..r_2)) > 0) \wedge \\ ((r_1 = r_2) \implies \max(\text{squareType}[r_1, j] \mid j \in \text{int}(c_1..c_2)) > 0) \end{aligned} \quad (2)$$

A visualisation of this whole mechanism can be seen in the figure 1.

### 4.2 Every Empty Square Must be Attacked

The puzzle requires that every square is threaten by a rook. To satisfy this condition we can go through every square and check their neighbourhood in row/column till an edge point (a block square, a clue square or the edge of the matrix) and force a possible rook in these neighbourhood.

We can divide the neighbourhood by row and column and try to satisfy one of them least. We can do this by applying an **or** ( $\vee$ ) between row and column neighbourhood.

For each neighbourhood, we need to find the start and end positions. We can find this



		3		
	X			
	X			
	X			
		1		

Figure 1: We are checking the in between squares of two rooks and try to find a square which is not an empty square.

positions by starting from our initial  $[r, c]$  and iterate the wanted direction until we find the edge while we are preserving there is only empty squares in the middle. For the start position of the row neighbourhood for instance, we can find this value like in this equation 3. This equation gives the minimum value possible which the sum of `squareType` up to that point is 0 while satisfying the next left value is a block or non-existent.

$$\min(i | i \in (1..r), i = 1 \vee \text{squareType}[i - 1, c] > 0, \sum_{ii=i}^{\text{row}} (\text{squareType}[ii, col] = 0)) \quad (3)$$

We can find the maximum value of the row neighbourhood in the same column like in the equation 4.

$$\min(i | i \in (r..n), i = n \vee \text{squareType}[i + 1, c] > 0, \sum_{ii=row}^i (\text{squareType}[ii, col] = 0)) \quad (4)$$

We can find the min and max values for the other neighbourhood and use all 4 values like in the equation 5. The maximum value of the rooks in between must be more than 0 indicates there must be one rook at least and by using `or` we say either one of them must satisfied. When we say more than 0, in can also indicate two rooks but since we are constraining no rooks attacks each other, this possibility is already avoided.

$$(\max(\text{rooks}[i, col] | i \in (\min_r..max_r)) > 0) \vee (\max(\text{rooks}[row, j] | j \in (\min_c..max_c)) > 0) \quad (5)$$

A visualisation of this whole mechanism can be seen in the figure 2.

### 4.3 No rooks on Blocks/Clues

The puzzle requires that no rook replacement on blocks or clues. This is evidently simple constraint to apply. We can indicate that for every  $[r, c]$  if there's an existent block or

		3		
×	×	×	×	×
		×		
		×		
		1		

Figure 2: For an empty square (the one in the bottom of clue value of 3 in this example), we are checking min, max edges for the two possible directions (row/column neighbourhood) and try to put rooks to them to satisfy that empty spot to be attacked.

clue in `squareType` matrix, we can force or rooks matrix to have 0 on it. It can be seen in the equation 6.

$$(\text{squareType}[r, c] > 0) \implies (\text{rooks}[r, c] = 0) \quad (6)$$

#### 4.4 Clue Values Must be Satisfied

The puzzle has special blocking squares called clues which have a number on them to indicate they require that amount of rooks next to them. This is also a simple constraint to implement. we can check four directions of a clue square and make sum value of the rooks existence checks equal to the clue value given. This can be seen in the equation 7. By checking them if they are equal to 1, we also avoid the case of their undefined situation. If the rooks matrix value is on the edge and one next value isn't available it will be undefined and the boolean comparison will return 0.

$$\begin{aligned}
 (\text{squareType}[r, c] = 2) \implies & \\
 & (((\text{rooks}[r - 1, c] = 1) \\
 & + (\text{rooks}[r + 1, c] = 1) \\
 & + (\text{rooks}[r, c - 1] = 1) \\
 & + (\text{rooks}[r, c + 1] = 1)) = \text{clue}[r, c]
 \end{aligned} \quad (7)$$

## 5 Evaluation

Using different sizes of the board we tested our model to understand the capabilities of our model. Our model can find the correct solutions for each given board without violating any rules of the puzzle. We used 4 different board sizes;

- tiny board of  $3 \times 3$  (check figure 3 or .param file for an example)
- medium board of  $5 \times 5$  (check figure 4 or .param file for an example)

- big board of  $10 \times 10$  (check .param file for an example)
- huge board of  $20 \times 20$  (check .param file for an example)



Figure 3: The tiny size board instance.

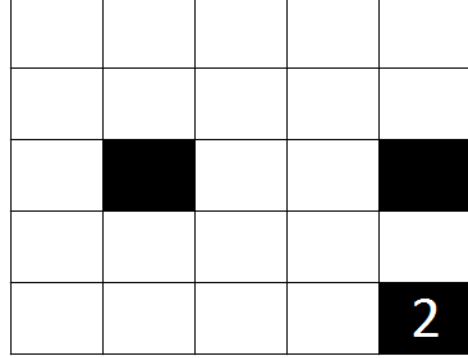


Figure 4: The medium size board instance.

The tiny and medium boards are relatively simple compare to two other since the wanted output value size really small. Additionally, when we introduce a block or clue we already reduce the decision domain of the output by eliminating them to have a value of 1. For instance, in the tiny board, 1 block reduces the the search size directly by one and makes it 8 which was 9.

Finding one solution only in tiny or medium boards take only 4 nodes (in savilerow) to use and really fast to solve. If the want every possible solution, a medium board requires 16 nodes to go though all the solutions.

The challenge start with big boards. A big board instance we created which we added with lots of blocks and clues take 13 nodes to find a solution. Additionally, finding all the possible solutions which the size of 336 is takes around 1200 nodes in this instance. And this computation with default optimisation took around 1.5 second to calculate which is acceptable.

The level of difficulty in the challenge can be increased in huge boards. An instance we created with significant number of blocks and clues takes 1276 nodes to find one single solution. This instance takes up to 15 seconds to translate and solve. This can be discussed with the fact that we chose to use an occurrence representation and because of that the model searches for 400 output value depending the constraints. The solving part actually takes less than 0.1 seconds so, we can point out the translation of the essence prime variables to minion takes most of the time which also indicates the I/O time requirements for the variables and the constraints which are translated to minion. All these solutions can be seen in the table 1.

If we try to find every solution in a huge board, it practically takes forever and we can't get the answer of it. So, in these, we can define our huge board as the challenge limit to find all solutions. However, finding limited amount of solutions in even bigger boards is possible.

We can test our model by testing in some examples. For doing this, we used the example

Instance	No Solutions	Node Size
model_tiny	2	6
model_medium	35	116
model_big	336	1288
model_huge	-	-

Table 1: The tests on different board size instances to find all the solutions.



Figure 5: The example instance which has been given in the specification of the problem.

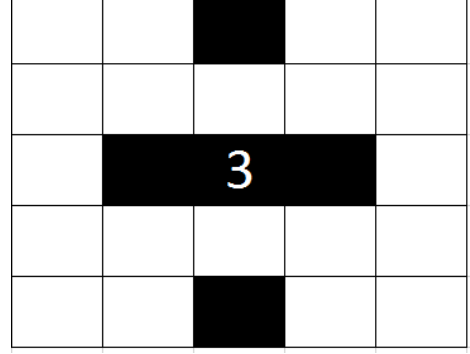


Figure 6: The impossible instance which contradicts with itself, therefore it has no solution.



Figure 7: The instance where everything is blocks with only one solution which is no rooks.

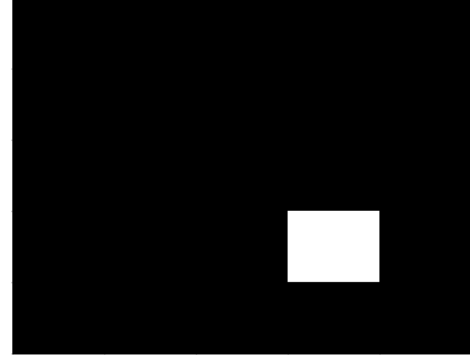


Figure 8: The instance where everything except one is blocks with only one solution which is one rook on the empty square.

case and created some other examples in medium size. This instances can be seen in the figures 5 6 7 8.

The results of these tests can be seen in the table 2.

Instance	No Solutions	Node Size
model.example	6	27
model.impossible	0	0
model.all	1	1
model.all_but_one	1	1

Table 2: The tests on using some example.

## 5.1 Optimisation Parameters

By using the optimisation options, savilerow tries to reduce the constraint size by eliminating some of the constraints by using different subexpression deductions. For our huge board instance with one solution finding mode, we can see the experiment results in the table 3. Instead of computer lab machines, we used our own computer to create and solve the models, so these time values may differ in other computers while the node size theoretically should stay the same.

Optimisation level	SavileRow Time (s)	Node Size	Minion Time (s)
O3	14.374	~ 1200	0.052
O2	14.048	~ 1200	0.056
O1	13.376	~ 1200	0.08
O0	11.866	~ 1200	0.112

Table 3: The optimisation level test using our huge board instance.

This experiment results show that actually optimisation increases the total time to find a solution. It's actually increases the computation time savilerow does to find subexpression deductions. It helps minion to take less time to solve the translated model. If we had a problem which requires a significant amount of time to solve in minion, this optimisations can be vital to decrease the search space and time. This can occur when searching for all solutions or the best solution depending on the definition of the best solution. Since we can't access to all solutions on our huge board, we can try to find a specific solution. An example can be given as adding a minimising objective to our model which we are going to discuss in detail in extensions section. Briefly, if we try to find the less amount rooks for the given puzzle instance, we add a minimising constraint and run savilerow to translate to model and run minion to solve it. The results of this can be seen in the table 4.

Optimisation level	SavileRow Time (s)	Node Size	Minion Time (s)
O3	14.720	~ 30M	102.384
O2	14.548	~ 30M	105.704
O1	13.528	~ 30M	175.26
O0	12.029	~ 30M	412.596

Table 4: The optimisation level test using our huge board instance with minimising objective.

As we can see, using optimisation parameters increase the time of savilerow to translate the model to minion but in minion part, it helps a lot if there's an objective to satisfy which can require iterating through huge number nodes.

## 5.2 Heuristics

Essence Prime has a parameter to indicate use heuristics on given branching parameters. We can use our rooks matrix to branch on and apply the heuristics. We used the minimising objective solution finding instance to test the different heuristics and retrieved these

results which can be seen in the table 5. All the tests are done using without additional optimisation parameter which uses default optimisation level.

Heuristics	SavileRow Time (s)	Node Size	Minion Time (s)
none	13.459	$\sim 30M$	103.720
sdf	13.633	$\sim 30M$	155.124
static	13.364	$\sim 30M$	103.476
conflict	13.649	$\sim 30M$	85.312
srf	13.388	$\sim 30M$	237.396

Table 5: The test of different heuristics using our huge board instance with minimising objective.

In our tests, the fastest acquiring of the best solution is done by **conflict** heuristic. The **static** one and no heuristics give similar results as the second one. **sdf** and **srf** gave worse results compared to the others and especially **srf** took so much time to arrive at the wanted solution. These heuristics values show that conflict heuristics suits the best for our model. However, it doesn't necessarily mean that conflict will be better in every model. In our model it showed the best results.

## 6 Extensions

Addition to our basic model we did some improvements in our model to capture symmetry and break it. We also tried to find the best available solution which has the less rooks in it on given configuration.

### 6.1 Symmetry Breaking

For symmetry breaking purposes we used a couple of test boards. For clear view, we chose them to be medium in size. They are;

- Horizontal flip symmetry example (figure 9)
- Vertical flip symmetry example (figure 10)
- Hor/Ver flip symmetry at the same time example (figure 13)
- Row symmetry example (figure 11)
- Column symmetry example (figure 12)
- An empty board example which has all the symmetries (figure 14)

#### 6.1.1 Horizontal/Vertical Flip Symmetry

Horizontal or Vertical flip symmetry exists when the board is symmetrical from  $y$  or  $x$  axis respectively. We first need to detect the occurrences of these symmetries before breaking them to reduce the space of the solutions.

To detect them we can go through all the rows or columns until the mid point and



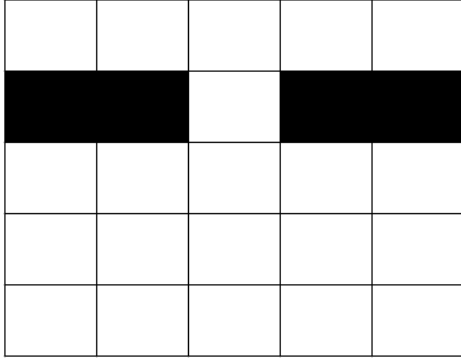


Figure 9: The horizontal symmetry instance.

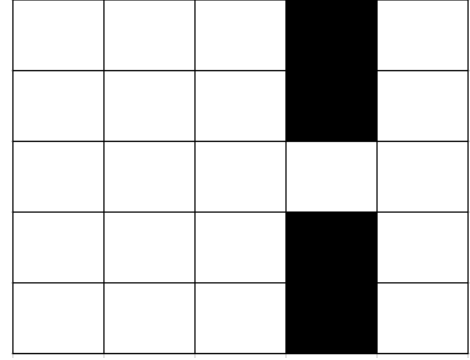


Figure 10: The vertical symmetry instance.

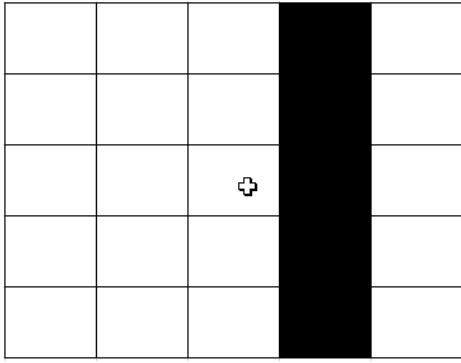


Figure 11: The row symmetry instance.

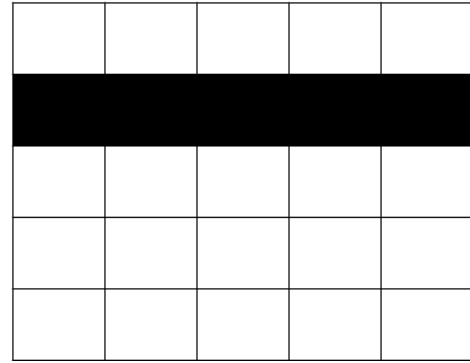


Figure 12: The column symmetry instance.

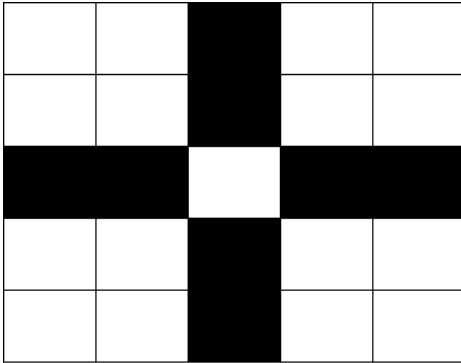


Figure 13: The hor/ver symmetry instance.

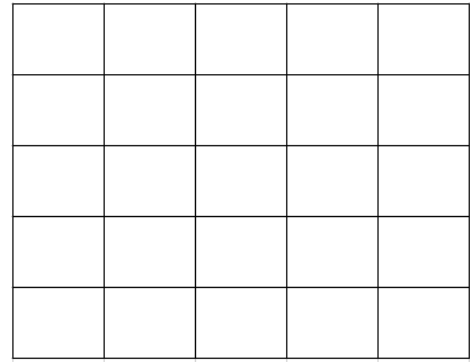


Figure 14: The empty board instance.

compare them with their complementary row or column. By complementary, we mean the symmetrical row or column by the  $x$  or  $y$  axis. For horizontal flip symmetry detecting for instance, we can go through the columns starting from 1 to  $n/2$  and compare `squareType[...c]` and `squareType[...n - c + 1]`. We can use *lex* ordering or we can simply check equality of each element in 2 columns. We also need to check the clues matrix to see if they have the same number written on them if they are clue. In the implementation, we subtracted the both values of `squareType` and `clues` in two complementary columns or rows and tried to make maximum of them to be less than 1 to satisfy this logic. It works as if we expected and we can detect the horizontal flip ( $y$  axis) and vertical flip ( $x$  axis) symmetry correctly. We store these informations as an output parameter in our

extended model to see in the solution file which symmetry is in the instance.

To break this symmetry we need to create submatrices of the rooks matrix depending on the available axis symmetry and by taking a reverse the second submatrix and order by lex. For instance, if vertical flip ( $x$  axis) symmetry is present in the instance, we can define our symmetry breaking like in the equation 8. This equation already takes the inverse of the second matrix since it iterates starting from the last row to the middle. We can apply the same logic to the horizontal flip symmetry by keeping rows same and iterating through the columns to the middle starting from 1 and  $n$  at the same time to the middle. When we compare these two matrices, we need to flatten them. Flattening is required since we can't just order 2d matrices so we put every value of the matrices to a 1d matrix and order those matrices by  $\geq$ lex.

$$\begin{aligned} &[rooks[i, j] | j \in (1..n), i \in (1..n/2)] \geq \\ &[rooks[n - i + 1, j] | j \in (1..n), i \in (1..n/2)] \end{aligned} \quad (8)$$

We can compare the results of the example instances using our basic and extended model. This can be seen in the table 6.

Model	instance	No Solutions	Node Size
basic	model_hor	264	794
	model_ver	264	882
	model_example	6	27
	model_vh	16	86
extended	model_hor	132	398
	model_ver	132	412
	model_example	3	14
	model_vh	9	52

Table 6: The test of different instances which have horizontal/vertical symmetry.

### 6.1.2 Row/Column Symmetry

Row/Column symmetry occurs when all the rows or columns are the same. Like we did for the flip symmetries we need to detect the row/column symmetry first to break it. We can do one iteration through rows or columns and check that row or column with the next one to see their equality. For row symmetry example, we can check every `squareType[r,c]` and `squareType[r+1,c]` and if they are not equal anywhere, we can max this value out from the matrix comprehension to say the symmetry doesn't exist. Otherwise, it will be indicate row symmetry. We also need to be careful about the clue matrix. Even they have the same value in `squareType` matrix the number written on them can be different that we need to check this equality as well if exist.

To break it we can simply lex order every row or column if we detect the symmetry. We need to use  $\geq$ lex instead of simple  $>$ lex because there can be some instances that two solutions rows or columns can be the same. Another thing we need to be careful about is the conflict with axis symmetry. Since the row/column symmetry also indicates axis symmetry, we need to make the ordering to be not conflicted and make the bigger

values in the same place for each symmetry breaking.

We can compare the results of the example instances using our basic and extended model. This can be seen in the table 7.

Model	instance	No Solutions	Node Size
basic	model_row	300	969
	model_col	300	785
	model_empty	120	1061
extended	model_row	4	8
	model_col	4	13
	model_empty	1	1

Table 7: The test of different instances which have row/column symmetry.

## 6.2 Minimising the Rook Size

The problem instance can also indicate an objective to be satisfied, We can indicate that we want the less amount of rooks to be placed in the puzzle. This solutions may have better importance in real life applications of this puzzle where the elements which rooks represent have costs to place.

We can minimise the total rook replacement by minimising the sum of the all rooks matrix value and minimise it. This will give the less number of rooks in the solution. We can check this situation in the example instance. The details can be seen in the table 8. The example situation after symmetry breaking has 3 solutions in which 2 of them has 7

Model	Node Size	No Solutions	Total time
extended	14	3	0.455
extended w/ objective	9	1	0.492

Table 8: The test of different heuristics using our huge board instance with minimising objective.

rooks and one of them has 6 rooks. By minimising objective, we achieve finding 6 rooks solution directly.

## 7 Conclusion

In the conclusion, we experimented using SavileRow with Minion and created a model with symmetry breaking extension to reduce the time/space complexity. We experimented creating boolean expression constraints using matrix comprehensions on available matrices. We evaluated our results and compared the effect of the symmetry breaking in constraint modelling.