Assignment 1: The Rook Placement Puzzle

A constraint model was written in Essence Prime to tackle the rook placement puzzle detailed in the practical specification.

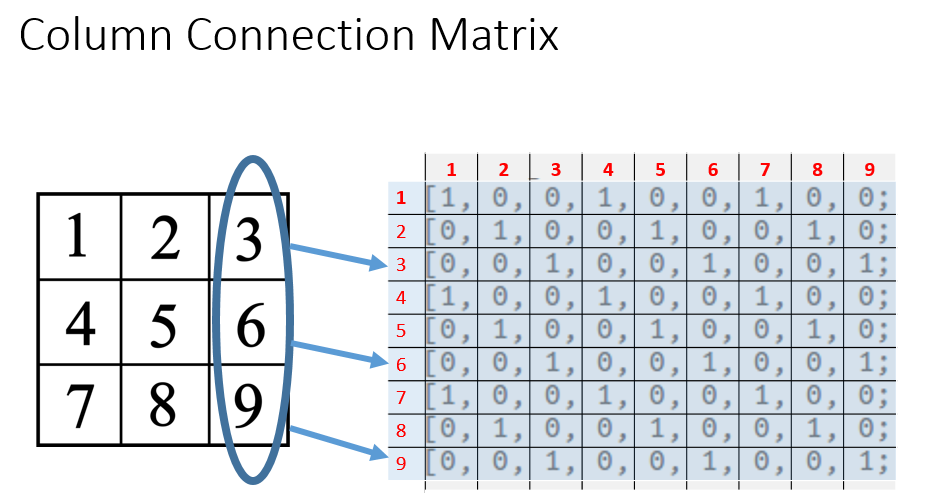
# Implementation

First of all, I convert the 2D squareType array into a one dimensional array which contains a 1 for squares with no blocking square and a 1 for blocking squares (including clues). The key part of the code to translate a position in a 2d array to a 1d array is the following formula where n is the number of squares in each row/column:

2DArray[row, col] = 1DArray[(n\*row)+col]

Then I create a connection matrix. To do this, I create a 25x25 matrix (n^2 by n^2), for a 3x3 chess board it would be a 9x9 matrix therefore. I decompose this problem into two parts. First of all, I create a matrix which gives a value of 1 for each square that is in the same column. To determine if a square is in the same column as another the modulus function is used. If two squares have the same remainder when divided by n, then they are determined to be in the same column. If they have a different remainder they are in a different column.

Each row and each column of this matrix corresponds to a square in the game board, so if it were a 3 by 3 matrix the column connection matrix would be as below:



Notice how the rows are the same for each of square 3, 6 and 9. For squares within the same column, the rows of the connection matrix will be equal for those squares. To read the matrix, if one wanted to see which squares where in the same column as the third square, one would look at the third row and the 1s represent squares in the same column. Alternatively one could look at the 3rd column of the matrix as the rows are equal to the columns.

A row connection matrix was then constructed as above, but instead of using the modulus function, it was checked that if two squares returned the same result when dividing by n, then they are in the same row. For example, in a 3x3 matrix, 4/3 returns 1 and 5/3 returns 1 so 4 and 5 are in the same row. The matrix is indexed from 0.

## Modelling blocked off squares in same column and Row

Then I modelled how the blocked off squares impacted these connection matrices. First of all, for the columns, the 1D blocking squares matrix is iterated over. If there is a blocking square, squares which are in the same column but higher in value than the blocking square are determined to be blocked off from squares in the same column as the blocking square but with a lower value. For example, for a 3x3 board, if a blocking square is in square 6, square 9 is blocked off from square 3). Therefore in this example, in the column blocked-off square connection matrix, there would be a value of 1 in [3,9] and [9,3]. If there were no other blocking squares, all the other values would be assigned a 0. To assign the zeros, if all the following for three squares (x,y,z) do not hold true then they are assigned zero:

1. There is a blocking square in y (any square between x and z)
2. Y is in the same column as x
3. Y is in the same column as z

This would assign a value of zero in the column blocked-off square connection matrix for [x,z] and [z,x]. This process is repeated to model squares blocked off from each other in the same row to get a row blocked-off square connection matrix.

## Combining the matrices

Then these matrices created above are combined to get a complete connection matrix. First of all, I define when the connection matrix will have a value of zero. If two squares [x,y] are not in the same column or row, then there will be a zero for [x,y] in each of the two blocked-off square connection matrices outlined above, in the combined blocked off squares connection matrix, and in the combined connection matrix. Details of what these combined matrices are described below.

1. **Blocked off squares connection matrix**

For this matrix, for index [x,y] if the column blocked-off square connection matrix contains 1 at [x,y] or the row blocked-off square connection matrix contains a 1 at [x,y] then this matrix has a 1 at [x,y]. If both blocked-off square connection matrices contain a zero at [x,y] then the blocked-off square connection matrix will be zero at [x,y].

1. **Complete Connection Matrix**

The complete connection matrix is constructed using the same-row and same-column connection matrices outlined above, and the blocked-off squares connection matrix.

Firstly, if two squares [x,y] are in the same column or in the same row (InSameRowMatrix[x,y] =1 or InSameColumnMatrix[x,y] =1) , and the blocked-off square connection matrix is zero at [x,y] (BlockedOffSquares[x,y] =0), then the connection matrix has a value of 1 at [x,y].

Secondly, if the two squares [x,y] are not in the same column and not in the row (InSameRowMatrix[x,y] =0 and InSameColumnMatrix[x,y] =0), then the connection matrix has a value of 0 at [x,y].

Thirdly, if two squares [x,y] are 1 in the blocked off squares connection matrix (BlockedOffSquares[x,y] =1) then the connection matrix is a 0 at [x,y] as the two squares are in the same row or column but are blocked off from each other by a blocking square.

## Constraining the Problem

The above has not constrained the problem, as there is only one possible connection-matrix. I then proceeded to constrain the possible rook positions based on the connection matrix. The first constraint is below:

forAll x: int(0..n\*n-1).(BlockingSquares1D[x] =0) ->

((sum y : int(0..n\*n-1) . (ConnectionMatrix[y,x] =1 /\ Rooks1D[y] =1) ) > 0 ),

Firstly, for any square [x] that is not a blocking square or a clue, there needs to be at least 1 rook in a square connected to it (ConnectionMatrix[y,x] =1, where y can be any other board-square).

Secondly, I had the following constraint:

forAll x: int(0..n\*n-1).(BlockingSquares1D[x] =0 /\ Rooks1D[x] =1) ->

((sum y : int(0..n\*n-1) . ((y!=x) /\ ConnectionMatrix[y,x] =1 /\ Rooks1D[y] =1) ) = 0 ),

For any square that is not a blocking square and contains a rook, for all other connected squares (y!=x and ConnectionMatrix[y,x]=1), they must not contain a rook (Rooks1D[y] =0, for all the connected squares).

Thirdly, I add in the following constraint:

forAll row, col: int(0..n-1).

squareType[row,col] >0 -> Rooks1D [(n\*(row))+col]=0,

If the square is a blocking square or a clue, then it cannot contain a rook.

## Clues

Then clues were implemented. For any clue square on the board, the adjacent squares need to contain in total the number of rooks as the value of the clue square. In my implementation, a problem I found was that if it were on a perimeter square of the board, then checking four adjacent squares would be out of bounds, so it was repeated for squares in the top and bottom row and column (checking against three adjacent squares) and the corners of the board (checking two adjacent squares).

## Finally, Converting the 1D Rooks Matrix to a 2D

The solution above contains a 1D matrix of rook positions (containing 1s in the squares containing rooks). This is converted to a 2D matrix using the following formula:

forAll row, col: int(0..n-1).

Rooks[row, col] = Rooks1D[(n\*row)+col]

# Testing

The implementation was tested on many different configurations and worked for all of them, including a 10x10 configuration, configurations with almost all blocking squares, and configurations with blocking squares in configurations around the perimeter of the board. Every configuration worked correctly. I believe that the solution would work with any n x n board which would have a valid solution. Certain board configurations did not return a solution, but these were once with no solution as if a clue square has 4 and it is two squares away from a clue square which also has four for example, this would have no solutions.

The most challenging configurations were the larger ones, with it being run in a Vmware with limited disc space, it even failed to run with some larger configurations on occasion as it ran out of memory.

# Heuristics and Optimization Testing

Different heuristics and levels of optimization were tested on the default problem specified in the assignment specification. The results are displayed for the different optimizations below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Optimization | Total Solutions | Minion Solver Nodes | Minion Solver total Time (s) | Minion Solver Time Out (s) | Savile Row Total Time (s) |
| None | 6 | 4 | 1.336 | 0 | 11.387 |
| Level one | 6 | 4 | .048 | 0 | 9.695 |
| Level two | 6 | 4 | .052 | 0 | 11.637 |
| Level 3 | 6 | 4 | .06 | 0 | 11.709 |
| Variable Deletion | 6 | 4 | .068 | 0 | 7.986 |
| No Mappers | 6 | 4 | 1.408 | 0 | 10.326 |
| Minion mappers | 6 | 4 | 1.428 | 0 | 9.734 |

The results for the different preprocessing options are displayed below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Preprocessing | Total Solutions | Minion Solver Nodes | Minion Solver total Time (s) | Minion Solver Time Out (s) | Savile Row Total Time (s) |
| No Preprocessing | 6 | 4 | .04 | 0 | 11.541 |
| GAC | 6 | 4 | .068 | 0 | 11.865 |
| SAC | 6 | 4 | .052 | 0 | 12.376 |
| SSAC | 6 | 4 | 6.096 | 0 | 17.096 |
| SAC Bounds | 6 | 4 | .048 | 0 | 11.445 |
| SSAC Bounds | 6 | 4 | 11.84 | 0 | 27.221 |

Overall, it was clear that variable deletion was an extremely effective optimization, followed by the level one optimization. Variable deletion removes decision variables when the variable is equal to a constant or another variable. This was effective for how the problem was modelled because in my implementation, I had many matrices which only had one possible solution (for modelling the connection matrix).

In terms of the preprocessing, they were ineffective, as the option with no preprocessing resulted in the lowest total time. It is interesting that SSAC and SSAC bounds were extremely ineffective. The preprocessing of these would have taking up a lot of time and would not have reduced search time enough to compensate. It would be interesting to see whether increasing the board size would make preprocessing more effective.

## Branching Heuristics

Branching was implemented on the rooks matrix (Rooks1D) and the different heuristics tested. Heuristics are useful as they can reduce the branching factor by essentially providing a means to guess whether it is on the correct path or on the incorrect path to find the solution.

The results are below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Branching Heuristic | Total Solutions | Minion Solver Nodes | Minion Solver total Time (s) | Minion Solver Time Out (s) | Savile Row Total Time (s) |
| No Heuristic | 6 | 4 | .04 | 0 | 11.541 |
| Static | 6 | 4 | .064 | 0 | 12.279 |
| Sdf | 6 | 4 | .052 | 0 | 12.663 |
| Conflict | 6 | 4 | .04 | 0 | 11.02 |
| Srf | 6 | 4 | .052 | 0 | 10.416 |

Overall, the srf heuristic had the lowest time, followed by the conflict heuristic and both the static and sdf heuristic were worse than using no branching heuristic. The srf heuristic has the following description in the Minion Manual:

“smallest ratio first, chooses unassigned variable with smallest percentage of its initial values remaining, break ties lexicographically”

Essentially, it is instantiating the variables with the smallest domain space of future values first, and this is successful because it reduces the branching for variables with larger domain spaces.

# Extension: Symmetry Breaking

## First Symmetry Breaking Implementation

For the first implementation of symmetry breaking, I used the following code:

forAll x: int(0..n-1).

(sum k : int(0..((n/2)-1)).(Rooks1D[(n\*x)+k] =1)\*k) <= (sum k : int((((n-1)/2))..(n-1)) .(Rooks1D[(n\*x)+k] =1)\*(n-1-k)),

This following code assigns a value to a rook based on its position, so that if x were 2, it would check (2n+k) for rooks for all values of k, assign a score to the square based on its position by multiplying by k and then take the sum of the scores:

(sum k : int(0..(((n-1)/2))).(Rooks1D[(n\*x)+k] =1)\*k)

So in a 5x5 board n would be 5 and 2n would be 10 and it would check the squares between the values of k of 0 to 2, it would check the sum of the following squares:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

If either of these squares had a rook in it, then it would be assigned the value k (square 10 would be assigned 0 and square 11 would be assigned a value of 1, square 12 would be 2).

The following code:

(sum k : int((((n-1)/2))..(n-1)) .(Rooks1D[(n\*x)+k] =1)\*(n-1-k))

This assigns a value of a rook based on its position from the right hand side of the board so that if x were 2, it would check the squares between (2n+k) for rooks for the values of k between (2 and 4), squares 12-14:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 |
| 10 | 11 | 12 | 13 | 14 |
| 15 | 16 | 17 | 18 | 19 |
| 20 | 21 | 22 | 23 | 24 |

Square 14 would be assigned 0 and square 12 would be assigned 2.

The constraint ensures that for each row, the sum of rooks in the right hand side is greater than or equal to the left hand side of each row, and this eliminates solutions. A non-valid solution for the third row would be if a rook were in 11, one could not have a rook in square 14 (assuming there is a blocking square between them so it is a valid solution).

This worked effectively to decrease the solution set to one solution and decrease the solving time. However, I realized that it does not check overall symmetry but just checks one row at a time, and it would likely eliminate some in symmetrical solutions and might lead to no solutions in certain circumstances. Therefore I decided to improve upon this constraint.

## Second Symmetry Breaking Implementation

To break a symmetry one needs to first of all test that the symmetry exists. First of all, I would take the left hand of the matrix. A transformation would be needed I determined to match it up with the right hand side of the matrix similarly to how I did in the above symmetry breaking.

To do this I realized that for every row in the top matrix, they could be mapped to the horizontally symmetrical matrix with a simple formula. A square[x,y] would become [n-1-x,y] in the second matrix. So square [0,0] would become [4,0].

For the first step, I combined the rooks solution matrix with the square-type and clue matrix, so that the solution matrix contained a 1 for a rook, 0 for empty square and a 2 for a clue square or blocking square (I decided not to differentiate between a clue and blocking square in terms of symmetry). Then new matrices were constructed from the solution board matrix by flipping it horizontally, vertically, diagonally along both axis and rotating it 90 degrees, 180 degrees and 270 degrees. Each of these was constructed using a simple formula like above to transpose the matrix.

Then for each type of symmetry I implemented a Lex-constraint so that the flattened solution board had to be lexographically greater than or equal to the flattened flipped and rotated solution boards.

## Testing the Symmetry Breaking

I tested the model with different symmetry breaking enabled for the configuration specified in the practical instructions and the results are below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Symmetry Breaking Option | Total Solutions | Minion Solver Nodes | Minion Solver total Time (s) | Minion Solver Time Out (s) | Savile Row Total Time (s) |
| None | 6 | 4 | 0.076 | 0 | 12.086 |
| First Implementation | 1 | 1 | 0.048 | 0 | 12.472 |
| Horizontal and Vertical | 3 | 2 | .056 | 0 | 16.043 |
| Diagonal | 4 | 3 | .052 | 0 | 12.692 |
| Flip symmetry along all axis | 1 | 1 | .052 | 0 | 16.542 |
| Only rotational symmetry | 2 | 2 | .052 | 0 | 12.306 |
| All flip and rotational Symmetry | 1 | 1 | .048 | 0 | 21.682 |

The symmetry breaking reduced the minion solver total time as it reduced the number of nodes, but the Saville row total time seemed to actually increase for some of the symmetry breaking. It reduced the number of solutions effectively and the nodes. The best symmetry breaking seemed to be the first implementation I did, although it would be interesting to test it with other configurations as well given more time.

## Extension: A different Constraint Model

As an extension I attempted to different constraint models. However, neither of them worked. One of them I modelled in terms of each row and column, there is a lower and upper limit to the number of rooks (Attempt 1). The upper limit for each row/column set as the number of blocking squares in that row/column. The lower limit was set as 1. This model did not work however, although time permitting this would have been likely possible.

In another attempt (Attempt2.eprime), I tried to model it so that each rook has a one dimensional matrix with the positions it is attacking. Then I could constrain it that a rook cannot have more than one of the squares it is attacking in any single other rook’s attack matrix. Then I attempted to add the constraint that at least one rook is attacking any square on the board. Finally, there was a constraint that if a rook was on a square, than it would have all the vertically and horizontally connected squares in its attack matrix, using gcc. If it isn’t in the same column or row as a square, than that square is not in its attack matrix. Again, this solution did not work due to time constraints.