CS4402: Practical 1
Constraint Modelling

Contents

List of Figures

List of Tables

I. Introduction

The purpose of this paper is to discuss the implementation and some empirical details of a constraint model applied to solve the N-Rooks. The N-Rooks problem describes a problem class in which some k rooks are placed on a $n \times n$ chess board, where none of the rooks can attack one another and every unoccupied square of the board has to be attacked. Furthermore there are blocking squares, which block of the rooks attacks from their position onwards, in these blocking squares there is a sub-class containing clues, which consist of an integer $i \in \{0, ..., 4\}$, that signifies that a given blocking square is to be surrounded by i rooks in the end, furthermore no rook can be placed on a blocking square. To solve the issue of what amount of rooks k to place on the board, and where to place them, we define a constraint model in Essence Prime to perform that task for us.

In doing so, the rest of this paper is structured as follows, II introduces our basic constraint model

and it's details, Section III contains empirical analysis of some of the constraint solving features, Section IV discusses possible symmetries inherent in the problem class, and finally, Section V concludes.

II. Basic Model

For our basic problem specification in Essence Prime we require three prespecified quantities, the dimensionality of the board n, a $n \times n$ matrix specifying the square types and positions, and a clue matrix detailing how many rooks are to surround a certain blocking square if it also is a clue square. The square type matrix is here encoded with elements $\in \{0, ..2\}$, where 0 signifies an empty square, 1 stands for a blocking square, and 2 indicates a blocking square that also contains a clue. The clue matrix is encoded with elements $\in \{0, ..., 4\}$ indicating the amount i of rooks to be placed around the given clue containing blocking square. Knowing this we search for a $n \times n$ solution matrix with elements $\in \{-1, ..., 2\}$, where a value of -1 conveys a rook at a given position.

The first constraint implemented in our model is rather simple, for any blocking square in the square type matrix, we also put a blocking square into our solution matrix, so that our initial problems and final solution are consistent. Besides this we also constrain our solution using a global cardinality condition stating that any any row and column not containing any blocking squares have to contain one rook and only empty squares otherwise. Although this constraint is not necessarily an absolute for every problem case, as cases in which this constraint is not valid for all solutions can be constructed, it did in practice however alleviate the burden of the constraint solver by a considerable amount, but can be taken out if so desired, more on this in Section III. Besides this we have a constraint specifying that each free square needs to be attacked by at least one rook, and that each rook cannot be attacked by another rook. This is done by checking these conditions between each edge and block, and group of blocks separated by free squares, where the actual check is done via a checksum based approach on each group uninterrupted group of free squares, starting either at any free square or rook. Besides this a checker for the clue squares was implemented , where an auxiliary vector of dimensions $1 \times (n+1)n + (n*2)$ recording the position of every rook is used to ensure that there is the correct amount of i rooks around each clue square. The dimensions of the auxiliary vector may seem somewhat arbitrary, they are however simply chosen as such to ensure to out-of-bounds referencing takes place during the computation of the solution.

II.I Prior Implementation

Aside from the above implementation, we also utilized a constraint model running over two Essence Prime files in a prior implementation, which was due to the fact that the atleast and atmost functions where rather heavily utilized in said implementation, and these functions do not allow decision variables as input for the minimum/ maximum occurrence of factors. In this implementation a fairly extensive counting scheme was utilized to assign minimum and maximum numbers of rooks per row, as well as global cardinality conditions for non blocked rows and columns. Although this implementation worked very well for some examples, such as the first example problem in III, it was rather poorly generalized, lengthy, and quite clumsily coded and as such was abandoned in favor of the much shorter more widely applicable implementation discussed above.

III. Empirical Analysis

In this section we investigate the empirical performance of our constraint model for a set of this N-Rooks problem settings, henceforth referred to as NR1-NR6, which can be found in Tables 5-10, where a solution found is shown, the initial setting can be easily derived from said solutions. In this

investigation we use three main measures of performance, the number of constraint solver nodes used to find the solutions, the time the solver needed to find these solutions, and the number of solutions found for a given problem. These metrics for our problems NR1-NR6 can be found in Table 1. In Table 1 we can see that the solver appears to perform adequately, but also that the computational cost drastically increases with the dimensionality of the problem. For NR5 and NR6 n was increased by 3 from the other problem, and the number of nodes used and solutions produced for these two problems increased exponentially, a rather obvious bad omen for any real world applications, but more on this in Section IV. Here the aforementioned global cardinality condition on all empty rows and columns can also come in handy, as we can see in Table 9, the solution found for NR5 leaves the last two columns empty, however, by including the gcc we can reduce the number of solver nodes used quite significantly from 5440 to 3535. This also means that some valid solutions can not be found by solver, or reproduced via symmetry for that matter, so the choice of the trade off between the number of nodes and the amount of admissible solutions that are lost will have to be made carefully.

Besides this we furthermore investigated the available in Savile Row, using the same metrics, however

Problem	Solver Nodes	Solver Time (ms)	Solutions Produced
NR1	18	43	6
NR2	19	42	8
NR3	6	37	81
NR4	5	38	4
NR5	663	116	6819
NR6	5440	268	4416

Table 1: Empirical results for our 6 problem settings, using the basic implementation

only NR5 and NR6 were reviewed here. The metrics for the four levels of optimisation can be found in Table 2, where we can see that overall, the optimisation method O3 seems to perform best, both when considering nodes and computational time, besides this the other three do not differ on the number nodes, but O0 appears clearly inferior when considering computational time. Considering this, in all solutions presented, outside of the ones in Table 2, optimisation 3 was used, to ensure peak performance. The number of solutions found was not affected by the optimisation chosen in our case.

Having found an optimal optimisation method, we next turn to finding a superior heuristic, should

Problem	Solver Nodes	Solver Time (ms)	Solutions Produced	Optimisation Level
NR5	792	357	6819	0
NR6	5464	940	4416	0
NR5	792	121	6819	1
NR6	5464	335	4416	1
NR5	680	120	6819	2
NR6	5464	311	4416	2
NR5	663	116	6819	3
NR6	5440	268	4416	3

Table 2: Empirical results for the four optimisation levels, problem sets NR5 and NR6

there be one, where once more the same metrics are used. The results for this investigation can be seen in Table 3, here, no clear winner can be found, although the conflict heuristic delivers a superior performance when it comes to the nodes opened up for NR6, it performs sub-optimally on NR5, and, in informal experiments, we have found that the conflict heuristic tends to open a high number of nodes on occasion. As for the other heuristics, there is only very little difference between their performance. As such, we chose to use the srf heuristic in all our experiments, sans the ones in Table ??, as it performed satisfactorily enough.

Problem	Solver Nodes	Solver Time (ms)	Solutions Produced	Heuristic
NR5	663	116	6819	sdf
NR6	5440	257	4416	sdf
NR5	663	118	6819	static
NR6	5440	257	4416	static
NR5	680	118	6819	conflict
NR6	5142	266	4416	conflict
NR5	663	116	6819	srf
NR6	5440	268	4416	srf

Table 3: Empirical results for our he four heuristics, problem settings NR5 and NR6

IV. Symmetry Breaking

Having covered the empirical analysis of the performance of our basic implementation, we next turn to an attempt at increasing the computational performance, symmetry breaking. As the chess board is a square, and blocking squares are rather often in horizontally or vertically symmetric positions, symmetry breaking is used to stop the solver from exploring solutions that are symmetric other valid solutions, and thus waste computational time. To do a rather simple symmetry breaking scheme was implemented, a lexicographical less-than-or-equal constraint placed was on the rows and columns of our board, given that the the given row and column pair was symmetric in blocking square position and clues. In the presence of clues, a rather simplistic scheme was introduced, where the halves of the board to be compared for symmetry were simply checked to have the same clue values, as matching clue blocks solely proved far more challenging than doing so for normal blocking squares. This implementation might pose a problem in eliminating all vertical or horizontal symmetry if more complex problem sets are run, or purposely used to show inefficiencies in the solver model, but it did not prove very problematic for our purposes and problem sets. The implementation of our model, excluding the empty row/ column gcc, implementing symmetry breaking, was then once more applied to our problem sets NR1-NR6, the results of which can be seen in Table 4. Table 4 clearly shows that using symmetry breaking can significantly reduce the number of solver nodes used for a given constraint problem, as well as decreasing the number of solutions produced for a given problem, where the lost solutions can of course be replicated by simply reapplying the symmetry to the found solutions.

Problem	Solver Nodes	Solver Time (ms)	Solutions Produced
NR1	3	39	2
NR2	7	36	2
NR3	6	40	18
NR4	1	33	1
NR5	600	106	3301
NR6	2355	160	1113

Table 4: Empirical results for our 6 problem settings, using symmetry breaking

V. Conclusion

In conclusion we found that a constraint model can be successfully used to solve the N-Rooks problem, we found that the fourth optimisation level in Savile Row, -O3, performed the best in our tests, while there no clearly superior heuristic method in our tests. Furthermore established that symmetry can be successfully used to limit solver nodes for the N-Rooks problems, and thus, to some degree, optimize

computational performance.

0	-1	2(3)	-1	0
0	0	-1	0	0
-1	1	0	1	-1
0	-1	0	0	0
0	0	2(1)	-1	0

Table 5: Problem NR1, encoding as described above, clue amount in brackets

0	-1	2(3)	-1	0
0	0	-1	0	0
-1	1	1	1	-1
0	-1	1	0	0
0	0	2(1)	-1	0

Table 6: Problem NR2, encoding as described above, clue amount in brackets

1	-1	0	0	1
-1	1	-1	1	-1
0	-1	1	-1	0
0	1	-1	1	0
1	-1	0	0	1

Table 7: Problem NR3, encoding as described above, clue amount in brackets

0	-1	2(3)	-1	0
0	0	-1	0	0
1	1	1	1	1
0	-1	0	0	0
0	0	2(0)	-1	0

Table 8: Problem NR4, encoding as described above, clue amount in brackets

-1	0	2(2)	-1	0	0	0	0
0	0	-1	0	0	0	0	0
0	-1	1	0	-1	0	1	-1
0	0	0	0	0	-1	0	0
0	0	0	0	1	0	-1	0
0	0	0	0	-1	0	0	0
0	0	-1	0	0	1	0	0
0	0	2(2)	0	0	-1	0	0

Table 9: Problem NR5, encoding as described above, clue amount in brackets

-1	0	2(2)	-1	0	0	0	0
0	0	-1	0	0	0	0	0
1	1	1	1	1	-1	0	0
-1	0	0	0	0	0	0	0
0	-1	0	0	0	0	0	0
0	0	0	0	-1	0	0	0
0	0	-1	0	0	0	0	0
0	0	2(2)	-1	0	0	0	0

Table 10: Problem NR6, encoding as described above, clue amount in brackets