CS4402 - Constraint Programming

Practical 1: Modelling the M-Queens Puzzle

Student number: 130018883 Words: 2983

---

1. A 0/1 Model of the M-Queens Puzzle

1.1 A 0/1 Model of the N-Queens Puzzle

I began by formulating a model for the N-Queens Puzzle using a 0/1 Model, which I would later adapt to form a model of the M-Queens Puzzle.

Here, the N-Queens Puzzle refers to placing N queens on an NxN chessboard so that no queen is attacking another. I use 'forward diagonal' to refer to a diagonal that goes from the top-left to the bottom-right, and 'backward diagonal' to refer to one that goes from the top-right to bottom-left. This is illustrated in Figure 1.

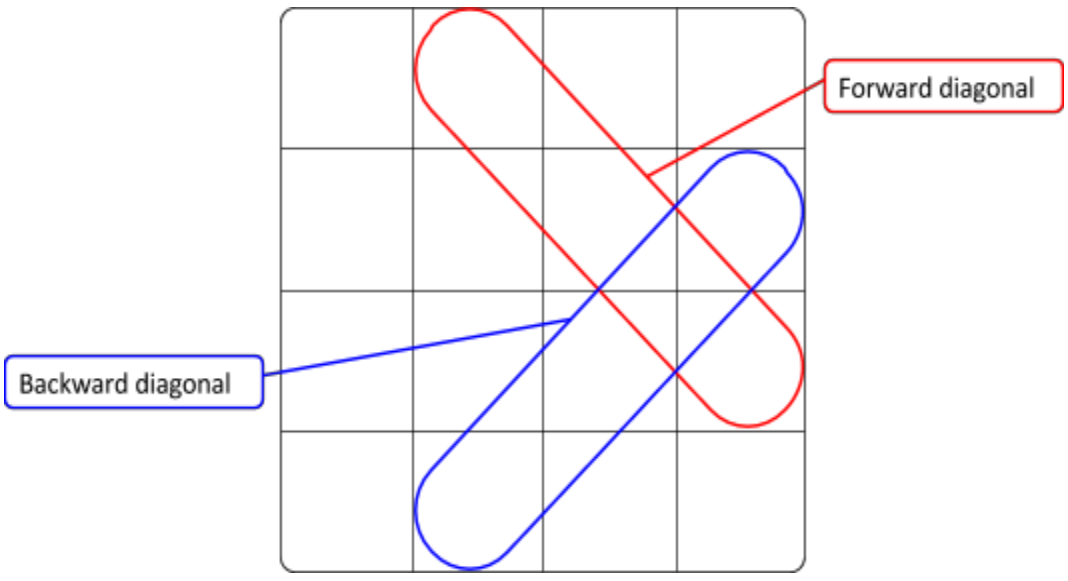

Figure 1

The constraints I used to achieve this were as follows:

- The sum of each row is equal to one, since there will be exactly one queen in each row
- The sum of each column is equal to one, similarly.
- The sum over each forward diagonal starting from the top row is at most one, so that no two queens attack each other along one of these.
- The sum over each forward diagonal starting from the first column is at most one, similarly, ignoring the main diagonal covered in the previous constraint
- The sum over each backward diagonal that starts from the top row is at most one, similarly.
- The sum over each backward diagonal that starts from the last column is at most one, similarly, and ignoring the main anti-diagonal covered in the previous constraint.

When forming the constraints for the diagonals, I began by summing over the forward diagonal for each element, and ensuring it was no more than one. I found this harder to encode for the backward diagonals, and on further analysis found that dividing these diagonals into two categories made things easier. One constraint covered diagonals whose uppermost cell was in the top row, and the other constraint covered diagonals whose uppermost cell was in the last column, as these two covered all cases. I also ensured the shared diagonal was only checked once, to minimize work done. After encoding this, when looking back at my model, I realised that the constraint I chose for forward diagonals was checking a lot of the diagonals multiple times, which was unnecessary. I used a similar process to the one I had used for backward diagonals and divided it into two constraints so that the sum of each forward diagonal was only checked once. I performed some tests for different values of N to confirm it was working, and found it to work as expected.

1.2 Adapting the N-Queens Model to an M-Queens Model

Once I had a working model for the N-Queens Puzzle, I made some changes so that it found solutions of the M-Queens Puzzle, as described in the practical description.

The first change was to alter the constraints that summed over columns and rows to be less than or equal to one, as opposed to equalling one exactly, as not all columns/rows necessarily contained a queen.

I then looked at adding a constraint that checked if a cell was being attacked. I added a constraint that ran over all cells, as each had to be checked. I used a disjunction of four separate clauses; one each to check if the cell was being attacked on a column, row, forward diagonal, or backward diagonal.

The clauses for checking columns and rows were relatively straightforward; I just checked that the sum over the row or column was equal to one for the given cell.

For checking the forward diagonal, I looked at the indices of the given cell, and tried to find a sum variable k so that I could sum over (i+k,j+k) to get the diagonal sum. I found that the uppermost cell, being always in either the first row or first column, had an index of 1 for i or j. I used min(i,j) to generalise the case, and find which index this would be, and then subtracted the return value from 1 to give the beginning point. Similarly, the lowermost cell would always be in either the nth column or row, so found max(i,j) and subtracted it from n to give the distance of the cell from the lower edge. By finding the beginning and end point for the sum variable k, I could then calculate the sum over the forward diagonal with (i+k,j+k) and set it equal to one for this clause.

Encoding the clause for checking if a cell was being attacked on a backward diagonal proved to be the most difficult challenge. The first thing I tried was using (i+j) as an identifier of the diagonal, as this quantity is constant across each backward diagonal, and is different for each diagonal. Similarly to the previous clause, I tried to use a mix of i, j, and n to form a sum over the diagonal, given I knew which diagonal the cell belonged to using the (i+j) quantity. I managed to get a convoluted expression, but this causes errors when run, which persisted after much troubleshooting, so I decided to scrap this approach.

I turned my attention to looking for a solution that used a sum variable k to sum (i-k,j+k) for the diagonal, which had worked for the backward diagonal constraint taken from the N-Queens Model. Using this template, I had to find the correct values to start and end the iteration of k. To see what this would look like I looked at the case M=4, and calculated two further 4x4 matrices: one each for the starting and ending index. The matrix I got for the starting index is shown in Figure 2.

0	-1	-2	-3
0	-1	-2	-2
0	-1	-1	-1
0	0	0	0

Figure 2

Trying to devise a function that returned these values for the given indices was tricky, but after some work I found that one way of getting this was by combining two other matrices, which are shown in Figure 3.

0	-1	-2	-3
0	-1	-2	-3
0	-1	-2	-3
0	-1	-2	-3

-3	-3	-3	-3
-2	-2	-2	-2
-1	-1	-1	-1
0	0	0	0

Figure 3

These two matrices can be gotten simply enough by calculating (1-j) and (n-i) respectively, and I used the max() functionality to get the combination, so that the sum variable starts from max(1-j,n-i). Similarly I got the end of the sum variable to be min(i-1,n-j), and so was able to encode the clause that checked for an attack on the backward diagonal.

Finally, in order to minimise the number of queens used, I added another variable for the total number of queens, added a constraint that this be equal to the sum over all cells, and added a minimising numQueens objective to the constraint model. After some final testing and fine tuning I found this model to work as desired for solving the M-Queens Puzzle.

2. A Different Viewpoint

2.1 A Coordinate model of the N-Queens Puzzle

Again, I began this part of the practical by trying to solve the N-Queens Puzzle, so that I could adapt the solution for the M-Queens Puzzle later. I decided on a viewpoint of an Nx2 matrix, where each row represents the x and y coordinate of a queen.

I ensured no two queens shared a row or column by applying an allDifferent constraint across the first and second column respectively. I identified the forward and backward diagonals by the difference and sum of the x and y coordinates respectively, and used nested forAll constraints over variables i and j to ensure they were not equal pairwise for i from 1 to n and j from i+1 to n.

As the rows of the solution matrix can be considered a set, this model did introduce a factor n! symmetry, however I got rid of this by enforcing an order on the entries in the first column.

2.2 Adapting the N-Queens Model to an M-Queens Model

The first issue to address when adapting this model for the M-Queens Puzzle was that I was looking for a set of non-fixed cardinality. Knowing that the number of queens had an upper bound of M, I added a variable numQueens that would represent the number of queens, and added a minimising numQueens constraint.

I began by adding a column to the coordinate matrix to act as a switch, where M[i,3] = 0 iff M[i,1] and M[1,2] were both equal 0, and M[i,3] = 1 otherwise. However, I realised that I was introducing far too many possibilities in the variable domain so instead opted for an Mx1 matrix called S for the switches, where S[i] = 0 iff M[i,1] = 0 and M[i,2] = 0. I summed over this switch matrix to get the numQueens. I also altered the ordering constraint to not strict, so that the (0,0) entries were pushed to the start, but it would accept multiple (0,0) entries.

Next I looked at changing the constraints on the first and second columns so that there didn't necessarily have to be a queen in each row and column. After coding some constraints that looked over all pairs of entries in a row, and having them be non-equal unless their entries were zero, I found the alldifferent_except() constraint in the Essence' manual, which served my purpose exactly. I used this to constrain these rows to be all different except if they were zero.

Finally I added a constraint to ensure every possible combination of coordinates was being attacked from at least one direction. I looked over all combinations of i and j, and used a disjunction of four clauses again; one for each direction. For each clause I used the exists constraint to ensure there was at least one case where the x coordinate was the same (share a row), the y coordinate was the same (share a column), the difference between the coordinates

was the same (share a forward diagonal), or the sum of the coordinates were the same (share a backward diagonal).

After some testing I realised that the (0,0) entries were validating cells along the main diagonal; i.e. where the difference between coordinates was equal zero. I altered this constraint to ignore the (0,0) cases, and this fixed the issue. I ran it for some different cases of M and found another issue where my constraint over the diagonals were preventing any queens being placed on the main diagonal. I played around with some different constraints, and decided on one that ran over i from 1 to n, and j from i+1 to n, which checked ensured that row i had a 0 entry before the diagonal constraint was enforced; i.e. the constraint is satisfied immediately if row i has dummy variables, but applies the constraint if not. After these final alterations the model worked satisfactorily for different values of M tested.

3. Empirical Evaluation

For this section, I ran both models for varying values of M, and tried branching on different variables with varying heuristics, and observing how this had an effect on the solution.

The first way of comparing the two models that came to mind was to run the solver for varying values of M and looking at the results. I looked specifically at the 'minion Nodes', 'minion TotalTime', and 'Savile Row TotalTime' details in the solution files, with the time taken to solve the problem being the most important. A table of these results can be seen in Figure 4 (below).

As can be seen, the 0/1 Model solved the problem much more quickly than the Coordinate Model for all values of M; the Coordinate Model taking 5 and 7 times longer respectively for M=10 and M=12. In general, it seems that this may be due to the model being less complex, and looks like it is related to the 0/1 Model generating less Minion Nodes, however for larger values of M this model has more Minion Nodes, and still solved the problem more quickly, so this does not necessarily hold.

Next, I took the 0/1 Model on its own, and looked at the effect branching on different variables with different heuristics had. I used a value of M=12 so that the disparities would be more evident. A table of the results can be seen in Figure 5 (below).

For branching on M, the number of Minion Nodes was consistent across all heuristics, except for 'conflict' where there were ~10% less. The total time taken to solve the problem didn't vary too much for different heuristics, with 'conflict' taking the least time, which seems to coincide with the lower number of nodes.

On the other hand, for branching on numQueens, the number of Minion Nodes was consistent for all heuristics, although noticeably larger than when branching on M. The time taken to solve the problem was roughly consistent across all heuristics, but again larger than when branching on M.

From these trials, I would conclude that branching on M is more efficient than branching on numQueens; the 'conflict' heuristic is slightly favourable for branching on M, and the choice of heuristic has no major effect when branching on numQueens.

I performed similar trials for the Coordinate Model, branching on the three different variables and changing the heuristic, again for M=12. These results are recorded in Figure 6.

When branching on M, making changes to the heuristic had a major effect; the 'conflict' heuristic resulted in the lowest number of Minion Nodes again, roughly ⅕ when compared to the 'static' heuristic. There was also a corresponding reduction in time, solving the problem much more quickly than when using the 'static' heuristic. Both the 'sdf' and 'srf' heuristics cause the number of nodes to increase, by a factor of ~2.5 in both cases. There was also a corresponding increase in time taken to solve the problem when using these two heuristics.

On the other hand, when branching on either S or numQueens making a change to the heuristic resulted in no change in the number of Minion Nodes. There was some change in time taken to solve the problem; the 'conflict' heuristic gave the shortest solution for branching on S, whereas all other heuristics had similar solution times; the 'sdf' heuristic gave the quickest solution for branching on numQueens, with 'static' being next quickest and the rest being roughly the same. The time taken longer than when branching on M for both 'static' and 'conflict' heuristics, but shorter when using both 'sdf' and 'srf'.

From these results, the optimal way to run the Coordinate Model is to branch on M and use the 'conflict' heuristic, as this gave a substantially lower number of Minion Nodes, and solution time. However, when compared to the trials run for the 0/1 Model, the solution time is still about twice as long, so I would conclude that the 0/1 Model is a better option for solving this puzzle.

Model:		**0/1 Model**	**Coordinate Model**
M = 4	minion nodes	27	39
	minion TotalTime	0.000999	0.011998
	Savile Row TotalTime	0.236	0.402
M = 6	minion nodes	413	506
	minion TotalTime	0.006998	0.6299
	Savile Row TotalTime	0.468	0.864
M = 8	minion nodes	9,267	10,434
	minion TotalTime	0.077988	0.561914
	Savile Row TotalTime	0.689	1.094
M = 10	minion nodes	122,319	92,347
	minion TotalTime	1.14083	5.73613
	Savile Row TotalTime	0.879	1.465
M = 12	minion nodes	11,476,889	10,949,641
	minion TotalTime	112.342	845.013
	Savile Row TotalTime	1.441	1.999

Figure 4

(M = 12)	Branching on:	M	numQueens
no heuristic	minion nodes	11,476,889	16,652,328
	minion TotalTime	112.543	161.712
	Savile Row TotalTime	1.091	1.089
heuristic static	minion nodes	11,476,889	16,652,328
	minion TotalTime	107.238	159.971
	Savile Row TotalTime	1.085	1.104
heuristic sdf	minion nodes	11,476,889	16,652,328
	minion TotalTime	113.702	162.091
	Savile Row TotalTime	1.083	1.086
heuristic conflict	minion nodes	10,374,034	16,652,328
	minion TotalTime	104.635	160.43
	Savile Row TotalTime	1.093	1.096
heuristic srf	minion nodes	11,476,889	16,652,328
	minion TotalTime	124.641	162.656
	Savile Row TotalTime	1.104	1.132

Figure 5

(M = 12)	Branching on:	M	S	numQueens
no heuristic	minion nodes	10,949,600	10,949,600	10,949,641
	minion TotalTime	846.1	879.493	876.845
	S-Row TotalTime	1.983	1.992	2.008
heuristic static	minion nodes	10,949,600	10,949,600	10,949,641
	minion TotalTime	845.764	875.645	853.022
	S-Row TotalTime	1.972	1.965	1.932
heuristic sdf	minion nodes	25,660,968	10,949,600	10,949,641
	minion TotalTime	1920.71	874.065	836.019
	S-Row TotalTime	1.989	1.981	1.942
heuristic conflict	minion nodes	2,193,801	10,949,600	10,949,641
	minion Time	202.901	842.615	888.145
	S-Row TotalTime	2.012	1.997	1.965
heuristic srf	minion nodes	25,660,968	10,949,600	10,949,641
	minion TotalTime	1888.09	879.187	887.145
	S-Row TotalTime	1.983	1.947	1.937

Figure 6

4: Symmetry

When looking at the symmetry, the first point that stood out was that solutionhood was preserved on rotating and reflecting the board. There are potentially 8 such solutions; the original board orientation, the board under rotation of 90°, 180°, and 270°, and their respective reflections. It should be noted that this does not necessarily mean there are 8 distinct solutions, as some subset of these 8 may coincide.

In order to see this in practice, I began by trying to run my 0/1 Model with the -all-solutions flag, but this was ignored due to the minimising constraint. To get a better idea I reverted to the N-Queens solution, and using the -all-solutions flag for different values of n, looked at how the solutions could be grouped into equivalence classes. I found that these coincided with the board rotation and reflection assumed in the previous paragraph.

I added some symmetry breaking constraints in order to try and remove some of these additional solutions, and only find one solution per equivalence class of rotations and reflections. I added 7 new variables of NxN matrices, and 7 constraints so that these would be set to the solution matrix under the rotations and reflections mentioned. I then added a constraint that the solution matrix was lexicographically less than or equal to each of these matrices (under flattening).

I added these constraints to my 0/1 Model for the M-Queens Puzzle, and compared it to the model without these symmetry breaking constraints for varying values of M. The results can be seen in Figure 7 (below).

In all cases, there is a smaller number of Minion Nodes, but the time taken in Savile Row is larger. I believe this is a result of the additional constraints; it takes longer for Savile Row to convert the model to one that can be passed into Minion, but results in a more simple model.

For small values of M, the solution time is shorter for the original model, although I believe that this is due to the simplicity of the puzzle. As M grows larger, the time needed by the original model increases more quickly than for the model with the symmetry breaking constraints, and for M=10 and 12 below the Model that breaks symmetry has a much shorter solution time.

Model:		**0/1 Model**	**0/1 + Symmetry**
M = 4	minion nodes	27	3
	minion TotalTime	0.000999	0.009998
	Savile Row TotalTime	0.236	0.482
M = 6	minion nodes	413	100
	minion TotalTime	0.006998	0.014997
	Savile Row TotalTime	0.468	0.795
M = 8	minion nodes	9,267	1,923
	minion TotalTime	0.077988	0.054991
	Savile Row TotalTime	0.689	1.074
M = 10	minion nodes	122,319	36,638
	minion TotalTime	1.14083	0.663899
	Savile Row TotalTime	0.879	1.274
M = 12	minion nodes	11,476,889	2,327,424
	minion TotalTime	112.342	40.6208
	Savile Row TotalTime	1.441	1.55

Figure 7