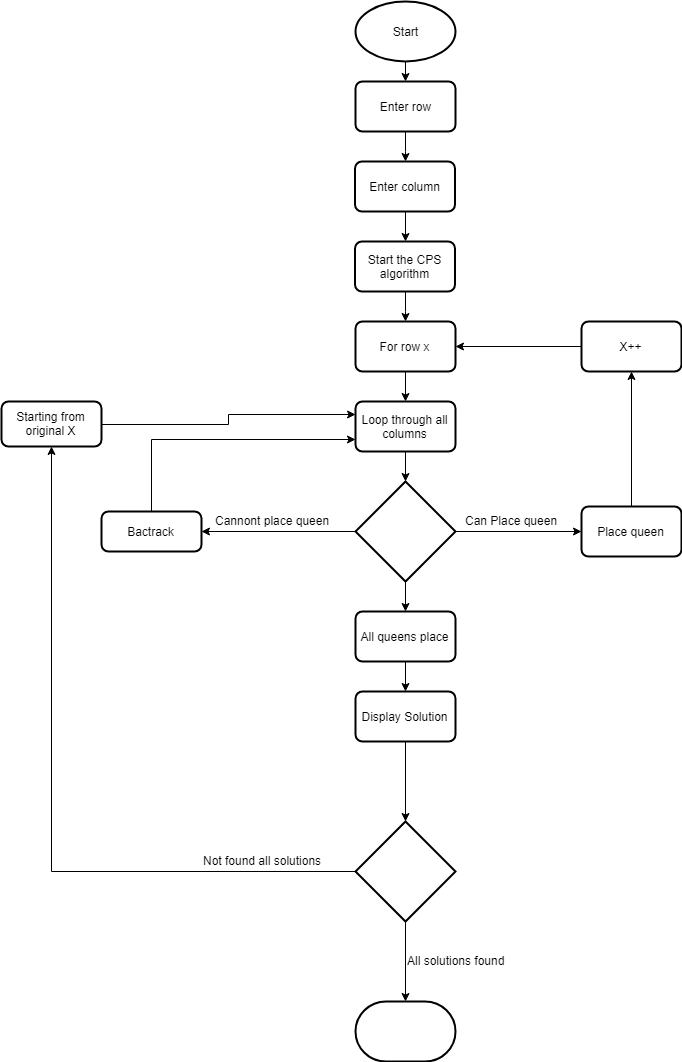
Warren Brown (15811541)

8-Queens problem

# CSP/Backtracking approach

I took a Constraint satisfaction problem approach (CSP) because this seemed the easiest and most efficient way of solving the 8 queens problem, as it is an N-Complete problem. Backtracking is one of the only ways to solve such a problem because it may require in some cases, all possible paths/choices to be taken/made to find a single solution depending on the placement of the user’s queen.

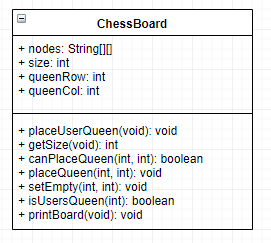
The CSP approach uses backtracking alongside a set of constraints that I have coded into a method, these constraints check if a potential queen position conflicts with any other queens that have already been placed on the board, such as horizontal, vertical and all the possible diagonal conflicts.



[Figure 1: Design]

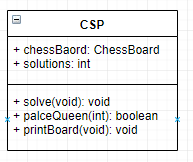
The design of the CSP was not difficult, as seen in the flowchart in Figure 1. The main feature of this CSP solution is that it uses a recursive method. This method takes in an integer as the starting row, it then goes through this row and calls the constraint method on all the columns. If it can place a queen then it places a queen, and then calls the recursive method from within itself to go to the next row and place the next queen, this happens for all the rows on the chessboard. The interesting thing to note is that if it cannot place a queen it returns false and ‘backtracks’ to the previous row, where it will then continue to check to check the remaining columns, and at first the backtracking part of the method took me a few days to get my head around and fully understand how it worked.

The most difficult part of this was designing it in a way that allowed for all the solutions to be found instead of it just returning the first solution that it found. This was done by using a Boolean variable within the method that would be set to the return of the method when called recursively, by doing this it searched for all possible solutions until the base case was triggered.



[Figure 2: ChessBoard class design]

It was important that I designed the chessboard in a way that allowed for it to be used by the CSP method. This is why I designed it to include the methods ‘canPlaceQueen()’, as it was better for this method to be contained within the ‘ChessBoard’ object rather than the CSP class as it meant that I didn’t have to pass in a node[][]/ChessBoard object into the method. This saved processing power and increased efficiency because potentially large node objects are now not being passed around the program.

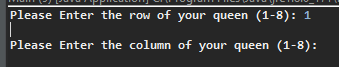


[Figure 3: CSP Class design]

The CSP class design was very simple as illustrated from Figure 3 the only things that the objected need to assess is a ChessBoard object and I also thought that I should keep a solutions counter. The methods are very clear, there are only two functional methods in this object and that is that solve and placeQueen method. The solve method is very simple and just calls the placeQueen method with the param ‘0’. The palceQueen method is the recursive function that solves the 8 queens problem, it does this by using the algorithm described in Figure 1.

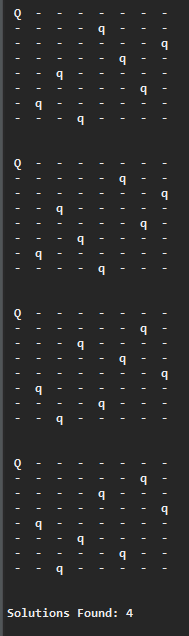
## Running the code

Running the code is very simple. The program asks for the user to enter a row and column between one and eight (Figure 4). Note that the row numbers start from the top of the board hence row 1 is the top and row 8 is the bottom row. The program will then place this queen and start to solve the problem.



[Figure 4: User entering queen position]

The solutions are then printed to the console as seen in Figure 5, along with the total solutions found at the bottom. I thought about storing all the solutions into an Array List but decided against this as the solutions only needed to be printed to screen and not used for anything else.



[Figure 5: produced solutions]