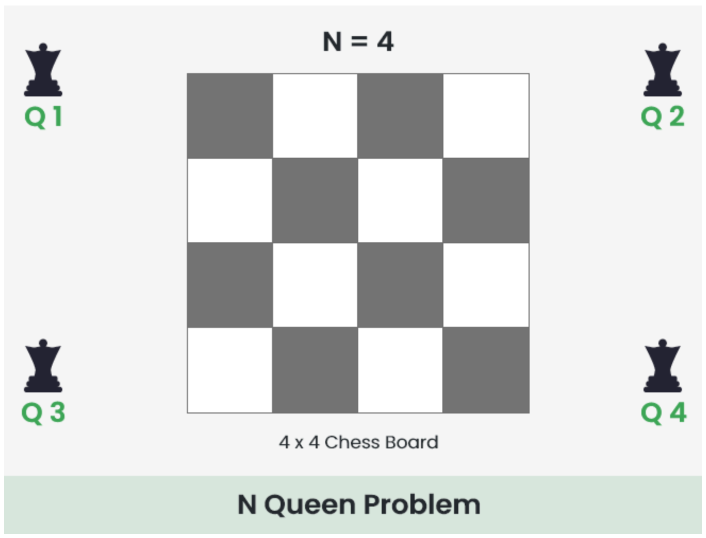
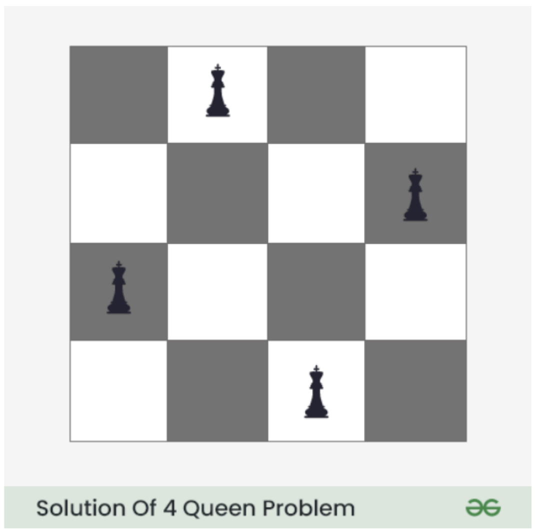
Names: Phillip Yan (pmy2105), Viktor Basharkevich (vb2574)

Description:





Source: https://www.geeksforgeeks.org/n-queen-problem-backtracking-3/

We will be attempting to parallelize the N-Queens problem, a classic combinatorial optimization problem in which the goal is to place N chess queens on a N by N chessboard so that no two queens threaten each other (i.e. are able to "take" one another). The input will be a given dimension N which will be both the number of queens and the dimension of the square board. The output will be the number of different combinations of queens that satisfy the conditions.

Our solution for this problem (unparalleled and single-threaded), is to use an implementation of depth first search. On every function call, the current matrix (an NxN array representing the board) is passed in, along with a row and column index.

Within every iteration/function call, we first check if the row/column index is out of bounds as our base case. If the base case hits, we return 0 if the current matrix is valid, and 0 if it is invalid. Validity is defined such that no queens are attacking each other (by checking that each row/column/diagonal only has 1 queen max), and that the matrix contains N total queens.

For the logic beyond the base case of the call, we have two options within every function: either add a queen at the current index we are on, or do not add a queen. Consequently, two function calls are made, incrementing the row/column indices, and passing in a new matrix that has either a new queen placed on it or not.

The nature of this algorithm means that for a board that is NxN, we are going to have a time complexity of O(2$^{\wedge N*N}$), because we have 2 branches of our depth first search, NxN times as we traverse through the checkerboard.

Note that within the single threaded algorithm, for each square which is eligible to have a queen placed, we call the depth first search algorithm twice; once if we decide to place the queen there and once if we do not. This branching factor is reminiscent of the branching situation in Fibonacci, which we parallelized in class. Thus, there is potential to parallelize these branches to optimize the aforementioned O(2$^{\wedge N*N}$) runtime.

Like in the Fibonacci, we will utilize the par monad to create a spark for one of the depth first search algorithms. That is, par (DFS$_{\text{includeQueen}}$) (DFS$_{\text{notIncludeQueen}}$). If there is an issue with creating too many sparks, we will experiment with threadscope to find optimal spark depth. As for getting input, that is trivial as the input is just N (size of the board).

When considering permutations of this problem, we could try and return the actual matrix representations of each valid board instead of the total number of valid boards. We could also remove the need for N queens as a constraint on the validity of a board.