Assignment 1

N-Queens Problem

CISC 352

Prof. Christian Muise

February 12, 2021

Jake Ahearne(20039207)

Muhammad Ahmed(20128941)

Brian Herman(20095996)

Sebastian Huber-Oikle(20049020)

Nick Johnson(20030070)

The Algorithm

Setting up the board (***set_board***)

Through the development and testing of the algorithm there were multiple approaches that were tested with the goal of maximizing efficiency and minimizing complexity. The first, a heuristic approach, returns false if the value for any n < 4. Otherwise, it populates the board assigning each queen, sequentially, in consecutive even-numbered rows. Afterwords, it populates the rest of the board with consecutive odd-numbered rows.

The next approach also dropped all given boards for every n < 4, the difference being that each queen is designated a unique row and column so that no queen will conflict with its neighbours, however, there can still be conflicts with queens that are further away. This approach weeds out many possible permutations of the board that would have easily avoidable conflicts. The number of board permutations that can be avoided by this approach increases exponentially with n for minimal additional runtime cost. The overall complexity of this approach is O(n).

We speculated that there would be no substantial increase in runtime to the min_conflicts algorithm. Using the previous function increases runtime due to the use of O(1) checks (if statements and comparisons). However, reducing the algorithm to a simple randomized board does not increase the average runtime of min_conflicts, while consistently increaseing the average runtime of set_board. Thus, using a randomized board will decrease the runtime of set_board, while not increasing the runtime of min_conflicts.

Iterative repair (***min_conflicts***)

After receiving an initialized board, the iterative repair first generates maps of positions on the board (row, positive diagonal, negative diagonal used to triangulate a cartesian point on the board), to individual queens and counts the number of queens related to each position, allowing us to access the domains of each queen in constant time. Originally, the steps of the algorithm found the queen with the largest number of conflicts and stored all the queens that it conflicted with and itself in a list. From this list, the possible rows for each of these queens were checked to find the row per queen that has the least number of conflicts. Then, the differences between the current number of conflicts per queen and the number of conflicts after moving were compared to find the queen that, when moved, would result in the overall highest difference in the total number of conflicts on the board.

We found that this solution costed substantially higher than required runtimes, and consistently ran into local minima. After completing this more complex method, it was speculated that a far simpler method may have a better runtime and may be able to avoid these frequent local minima. We then rewrote the function attempting to find a simpler solution that would be able to run faster (we tried one that more closely resembles the algorithm provided in the assignment).

In the rewritten function we use the same domain mapping as the previous method. We then generated a few data structures and variables to help track and access all the queens that initially had conflicts as well as the number of conflicts per queen. Then for each iteration we would choose a random queen from this set, find all the places within its column that have the lowest number of conflicts and randomly select one of these places to move to. This is done by

checking the corresponding values for each position with our domain data (when the position being checked is the same as our initial position, we must account for the presence of our own piece within the domain, thus –3 as our own queen will be in each of the 3 positions). If the row our chosen queen is moved to, has no conflicts, then the queen will be removed from the data structures that store the conflicting queens, if not, then the new number of conflicts for our queen is updated. Now the function updates the domain and board by removing the presence of the queen from its previous positions in the domain and adding it to its new positions. If the queen is moved to an initialized entry in our domain, we must check to see if there are any queens that it conflicts with that did not previously have any conflicts. These queens must be added to the set of conflicting queens for the next iteration to be able to choose them. Once the board is updated, if the queen that was moved, was moved to a place with no conflicts then we can check the board to see if it is a valid solution. If the board is valid then the function will return this board as a list, such that each index represents a column (0^{th} index = first column) and the value at each index represent the row of that queen. If the board is not valid, then increase the number of steps taken and repeat the process.

Solving (***solve***)

The first thing to note about this function is the unconventional approach to determining the number of maximum steps needed to solve the board. This value was determined by plotting the average steps required to solve boards of varying sizes of n. The plot was regressed to give the function for any given board size n. A more conventional approach like n * n or n * 100 could be used, though it is more amusing to derive such a convoluted algorithm that, though accurate, is mostly pointless. In the case of this algorithm, the occurrence of reaching max steps is next to none and the need to restart with a new board should not happen.

The solve function calls min_conflicts for n and the calculated maximum number of steps. While min_conflicts does not return a valid board, it will continue to call the function again until a valid board is found. Once a valid board is found, the board will be returned.

Checking for a Valid Configuration (***is_valid***)

The validation function reads the dictionaries that map a given position (row, positive diagonal, negative diagonal) on the board to the number of queens that share this attribute. The columns can be disregarded due to the nature of the data structure used, and thus no two queens can share an index/column. If the validation function determines that a position is populated by more than one queen, then the board is not a valid solution. If each position contains at most one queen, the function returns true indicating a valid solution.

Contributions:

Name	Code (%)	Document (%)
Jake Ahearne(20039207)	0	15
Muhammad Ahmed(20128941)	0	15
Brian Herman(20095996)	70	10
Nick Johnson(20030070)	28	60
Sebastian Huber-Oikle(20049020)	2	0