CS 203

Assignment 1

N Queens Problem

By:

Colin Quinn

Preview:

In order to execute these java programs, load each file into any preferred IDE then compile and run, a prompt in console will ask for input and give further instructions. The process is the same for both programs. Compiling from the command line should also be available if that is preferred. The focus of these results will be on the runtime of two algorithms that solve the N-queens problem. The first is the brute force approach that generates every permutation possible for the chess board until a solution is found, and the second is to generate a possible solution and attempt to swap queens until a solution is found. The times for this test were performed on an HP Pavilion with an Intel i3-7100U clocked at 2.4GHz.

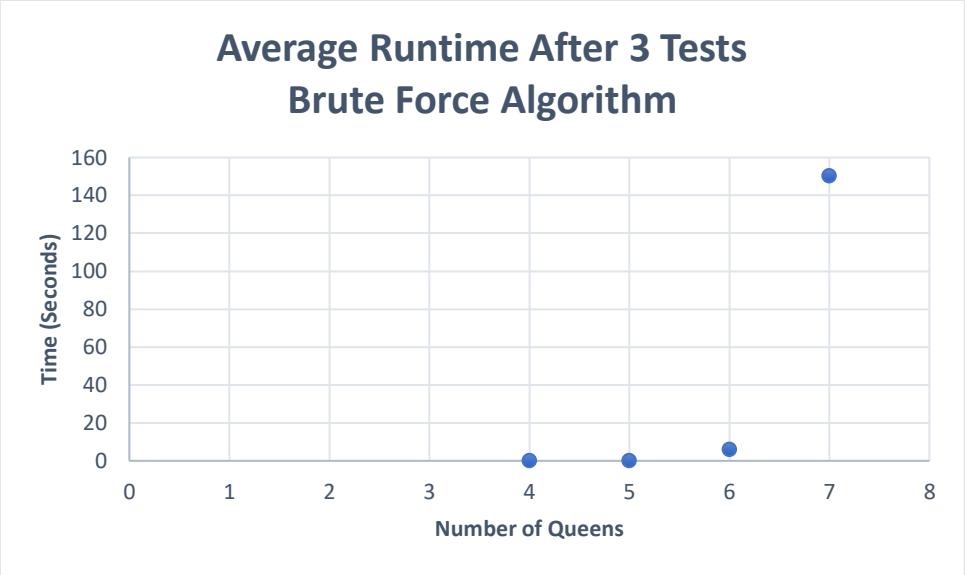
Brute Force Theoretical:

This algorithm has no defined function for efficiency due to its reliance on not just the size of the chess board or number of queens, but also contents of the chess board. Therefore, the algorithm requires a best case and a worst case for generating a solution. The best-case scenario for the brute force approach is that the first board generated is a solution to the problem, which would require placing the n amount of queens and checking the board's columns, rows, and all diagonals. Where checking rows and columns are both n^2, thus making it $2\ n^2$, as well as checking diagonals which implemented in the algorithm attached, is $4n^3 - 2\ n^2$. Making the overall check on a correct board require $8\ n^5 - 2\ n^2$ comparisons in the worst case, which means the board is a solution. The reason for this less efficient implementation to check diagonals is the outer loop that checks which diagonal is currently being evaluated, and then checking each square if it is in the diagonal and its value. In the worst case of the brute force algorithm, the final permutation generated is a solution. In order to generate and check all permutations, the algorithm must place all possible queens in all possible locations. This means that for board of n

size, there are n^2 possible locations for the first queen, $n^2 - 1$ possible locations for the second queen, and so on until $n^2 - (n + 1)$ is reached, in which case the recursion stops. The worst case for the algorithm then ends up requiring $((n^2)! / (n^2 - n)!)) \times (8n^5 - 2n^2)$ comparisons to find a valid solution. Therefore, the worst case for this implementation of a brute force approach is within $\theta((n^2)!)$ while the best case is in $\theta(n^5)$. This means that the brute force algorithm attached runs in $O((n^2)!)$ where n is dictated by the magnitude of input.

Brute Force Empirical:

The brute force approach yields rather lackluster results, but it is quite reasonable considering the amount of operations required. In the smallest case where $n = 4$, this algorithm requires 5216 different generations of boards before the first solution is found, and took on average over three executions 0.00399 seconds; while the largest case that was able to find a solution was $n = 7$ requiring 1,792,568,029 unique boards for a total time of 150.224 seconds. The reason for such a drastic change in time for such a small difference in number of queens is due to the classification of $O((n^2)!)$ where any small increase results in much more processing time. The limiting factor of this algorithm is the generation of every permutation through recursion. The case of $n = 8$ was unable to be computed on the test machine due to low memory and limited processing power, as the test was ran three times, the first for 54 minutes (3,240 seconds), then 83 minutes (4,980 seconds), and 71 minutes (4,260 seconds) before the application crashed.



Iterative Repair Theoretical:

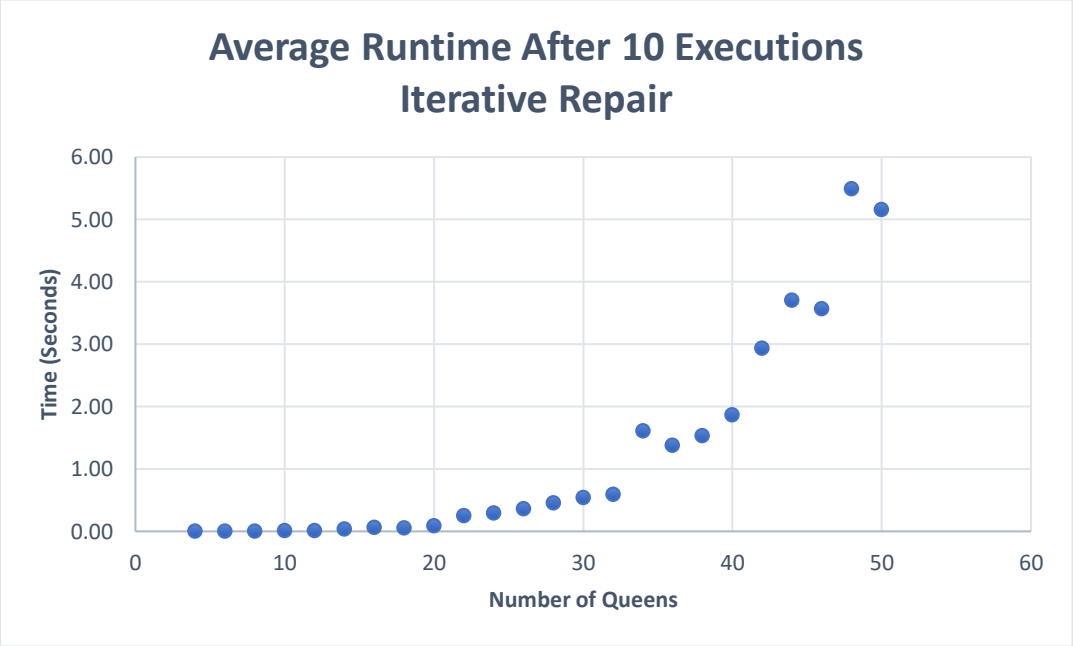
Due to the iterative repair algorithm's reliance on not just the contents of the input, the size of the input, but it also must implement some sort of shuffle feature in order to break out of local minimums. This shuffle is based on a randomly generated permutation where collisions only occur on diagonals, but because there are some permutations that cannot be solved by swapping queens, in the worst case the runtime is infinite. This theoretically infinite runtime would occur if the algorithm generates the same unsolvable permutation every time it is required to shuffle the layout of the queens. Therefore, the nature of randomness means that the efficiency of the rest of the algorithm can be ignored in the worst case as this randomness includes at least one permutation that cannot be solved using the iterative repair method. Meanwhile in the best case, it is possible that the first randomly generated permutation is a valid solution, so the only work required would be generating the board and validating that it is a solution. To initialize the board with a queen on each space of the main diagonal, n^2 comparisons are required to fill the chess board. Then in order to shuffle the chess board, n swaps are performed as the implemented shuffle method passes through the array once swapping contents of columns. This leaves the

work required to verify the solution which is done in checking the positive and negative diagonals of the board and calculating the conflicts. In order to find the number of conflicts per diagonal, the number of queens is added to an array of $2n - 1$ size where n^2 comparisons are required to find contents of each diagonal. Thus, in order to sum the queens on each diagonal, $4n^3 - 4n^2$ comparisons are required. Then in order to find conflicts, each element of the arrays must be compared again to find total number of conflicts, this requires $4n - 2$ comparisons to complete. Adding this work together, the best case for this implementation of iterative repair is $4n^3 - 3n^2 + 5n - 2$, which is classified as $\theta(n^3)$. Therefore, in the worst case the runtime for iterative repair is infinite, while the best case is $\theta(n^3)$, meaning the algorithm can be classified as $\Omega(n^3)$.

Iterative Repair Empirical:

Due to the implementation of randomizing the layout of the board if a local minimum is reached, averages over a larger sample are needed in order to get an accurate measure of the runtime. This was achieved by putting the main algorithm call in a loop and storing time per iteration of the algorithm in an array to calculate the average runtime. The case of $n = 4$ required an average of 0.000405 seconds after ten executions, while a case of $n = 50$ required on average just 5.1455 seconds. The main bottleneck of this algorithm is the check of the number of queens on each diagonal, since the way it was implemented requires n^3 comparisons, and a more

efficient diagonal check algorithm could improve the time required to find a solution.



Conclusions:

While the brute force approach is much more simple than iterative repair, the lack in efficiency holds the algorithm's effectiveness back drastically. While the runtimes are not bad for sizes $n = 4$ and $n = 5$, the brute force algorithm is essentially unusable for sizes any larger than those. The brute force algorithm was unable to find a solution after an average of a 74 minutes among three tests before crashing on the test computer. Meanwhile the iterative repair algorithm for $n = 50$ is comparable to the brute force's time for $n = 6$. Although the iterative repair has much more drastic differences in runtimes per board size due to the randomness required to always find a solution. Hence the need for an average time over a larger sample size in order to get a more accurate time needed for the iterative repair algorithm. The iterative repair is a much more in depth solution to this problem that does not rely on sheer processing power like brute force, but rather implementing smarter techniques to achieve the same goal.