Demetri Karras

Kettering University

CS-203: Computing and Algorithms III

Professor Giuseppe Turini

May 14th, 2025

Algorithm Description

The implementation of my algorithm closely follows the iterative repair algorithm designed by Rok Sosic and Jun Gu.

A chessboard configuration is randomly generated, with each queen having a unique column and row, effectively eliminating horizonal and vertical attacks.

A while-loop is ran while there are queen collisions on the board. In this loop, there is a for-loop nested in another for-loop. The outer for-loop iterates through the queens on the board. The inner loop iterates through queens on the board, but begins at +1 of the index of the outer for-loop. Therefore, the innermost loop has access to all possible combinations of queens.

If either of the two queens are attacked, a condition is evaluated to see if swapping them reduces the number of attacks. If it does, the swap is conducted. If not, nothing happens.

If no swaps were performed by the end of the iterations, the algorithm is stuck, and a new board is randomly generated.

Variables are maintained to keep count of the total number of swaps performed, the number of swap-round resets, the number of board reshuffles, and runtime.

A slight deviation from the provided algorithm is the condition for which the while-loop runs. In my algorithm, the algorithm runs while the total number of attacks is greater than zero. In the provided algorithm, the algorithm runs while the number of swaps on the current iteration is greater than zero. This means my algorithm performs more basic operations when evaluating the while-loop condition, which is negligible as the efficiency class remains the same either way. I also implemented a small code block that checks the number of attacks and current swap count to reshuffle the board, which appears different from the provided algorithms.

There are no bugs in my algorithm. There is a possibility for an infinite loop if boards without potential swaps that lower the attack count are continuously generated. This is extremely unlikely, as most boards have swaps that can be made.

Theoretical Analysis

Input Size: dimension of chessboard (n)

Basic Operations in Chessboard.java

- Chessboard(int dimension) – calls buildChessboard() and buildDiagonalArrays()
  - basic operations from helper methods – no loops
  - C(n) = 3n – 1 + n = 4n - 1
- buildChessboard()
  - adding an element to a list – in a for-loop
  - 1 instance of running Java's Collections.shuffle(List<?> list), which has n – 1 basic operations, with the input size being the size of the input list
  - element access and assignment – in a for-loop
  - C(n) = ∑[from i = 0 to n – 1](1) + (n – 1) + ∑[from i = 0 to n – 1](1) = 3n - 1
- buildDiagonalArrays()
  - element access, assignment, arithmetic, incrementing – in a for-loop
  - C(n) = ∑[from i = 0 to n – 1](1) = n
- swapQueenColumns(int queenRow1, int queenRow2)
  - all operations can be considered one basic operation – no loops
    - incrementing/decrementing, assignment, element access
  - C(n) = 1
- queenIsAttacks(int queenRow)
  - all operations can be considered one basic operation – no loops
    - assignment, condition evaluation, element access
  - C(n) = 1
- swapReducesAttacks(int queen1Row, int queen2Row)
  - all operations can be considered one basic operation – no loops
    - incrementing/decrementing, arithmetic, assignment, condition evaluation, element access
  - C(n) = 1
- getTotalAttacks()
  - condition evaluations and possible arithmetic – in a for-loop
  - C(n) = ∑[from i = 0 to ((2 * n – 1) – 1)](1) = 2n – 1

Best-Case

Basic Operations:

- Chessboard(int dimension) called at the beginning

- getTotalAttacks() called only once when evaluating the while-loop condition

The best-case scenario occurs when a solved board is generated immediately. The condition of the while-loop fails, and the main logic of the algorithm is never reached.

$C_{best}(n) = (4n - 1) + (2n - 1) = 6n - 2$

$6n - 2 \in \Theta(n)$

Worst-Case

Basic Operations:

- getTotalAttacks() called twice for every iteration of the while-loop
- queenIsAttacked(int queenRow) called twice for every iteration of the innermost for-loop
- swapReducesAttacks(int queen1Row, int queen2Row), swapQueenColumns(int queen1Row, int queen2Row), and basic arithmetic are done on certain conditions – all of which are constant time operations, so they can be grouped with queenIsAttacked(int queenRow) as one

The common worst-case occurs when each iteration of the while-loop only reduces the number of diagonal attacks by 1, while also never reaching a state in which the board needs to be reshuffled. Because any generated board can have a maximum of n – 1 diagonal attacks, the outermost while-loop can iterate for a maximum of n times.

$C_{gen.\ worst}(n) = \sum[\text{from } i = 0 \text{ to } n - 1](\sum[\text{from } j = 0 \text{ to } n - 1](\sum[\text{from } k = j + 1 \text{ to } n - 1](1)))$

$= \sum[\text{from } i = 0 \text{ to } n - 1](\sum[\text{from } j = 0 \text{ to } n - 1](n - 1 - (j + 1) + 1)$

$= \sum[\text{from } i = 0 \text{ to } n - 1](\sum[\text{from } j = 0 \text{ to } n - 1](n - j - 1)$

$= \sum[\text{from } i = 0 \text{ to } n - 1](n(n - 1) - (n(n - 1)) / 2)$

$= \sum[\text{from } i = 0 \text{ to } n - 1](n(n - 1)) / 2)$

$= (n^2(n - 1)) / 2 = n^3/2 - n^2/2$

$n^3/2 - n^2/2 \in \Theta(n^3)$

The absolute worst-case scenario is the algorithm running indefinitely. The algorithm gets "stuck" when no swaps were performed on an iteration. When this occurs, a new board is randomly generated. There is no guarantee that this new board will not get "stuck" as before, so there is a very unlikely chance that the algorithm will produce an infinite number of boards that cannot be solved using this method.

$C_{abs.\ worst}(n) \in \Theta(\infty)$

Most-Common Case

Because the absolute worst-case is so unlikely, the most-common case will fall somewhere between the best-case and the common worst-case.

$C_{\text{most common}}(n) \in \mathbf{O(n^3)}$, $C_{\text{most common}}(n) \in \mathbf{\Omega(n)}$

Graph

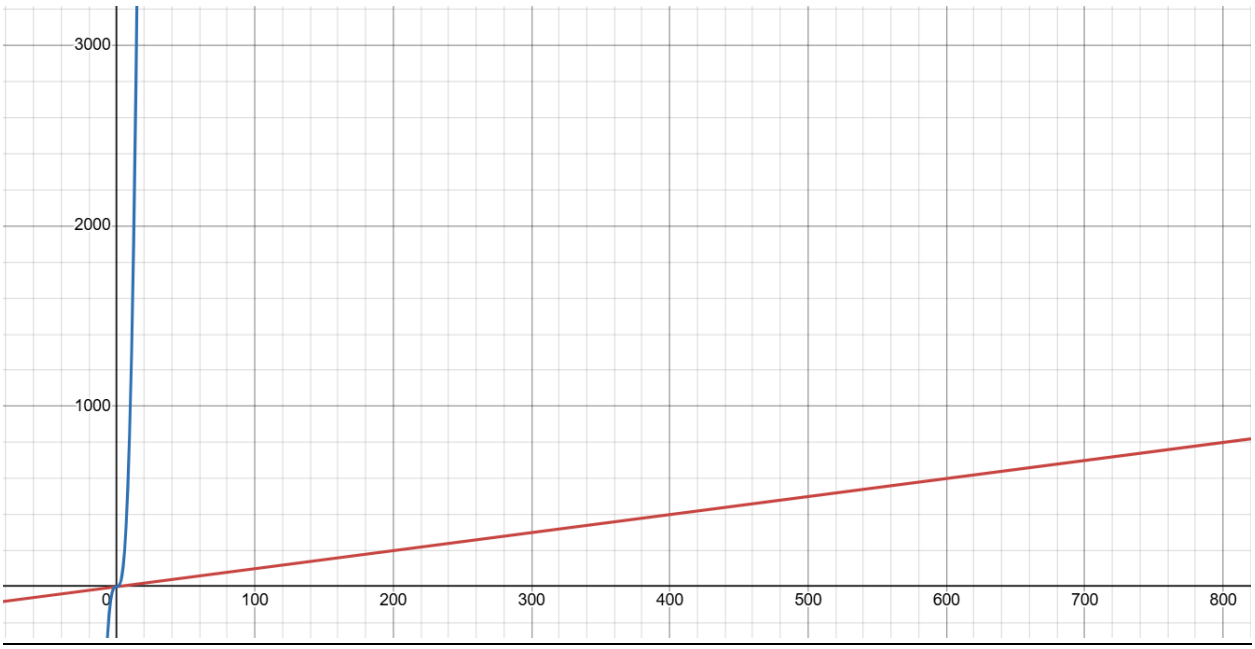
Red Line: $\Theta(n)$

Blue Line: $\Theta(n^3)$

Title: Theoretical Time Efficiency

X-axis: input size

Y-axis: time



The area between the red and blue lines represents the area where the line for the most-common case will fall.

Empirical Analysis

Hardware and Software

CPU: 13th Gen Intel(R) Core(TM) i7-1355U 1.70 GHz

RAM: 32 GB

Operating System: Windows 11 64-bit

Java Version: Java 23

Execution Tool: IntelliJ IDEA 2025.1.1.1

Description

To conduct this analysis, I ran 7 trials for 6 distinct input sizes. I chose the following input sizes to provide a wide range of relevant data: n = [10, 100, 1,000, 5,000, 10,000, 20,000]. Choosing values >100,000 made the data difficult to graph, and didn't provide as valuable insight. These input sizes also help illustrate the trend of the data best when graphed. I used Java's System.nanoTime method to time the algorithm's execution, and I converted the time into milliseconds. When collecting data, there were a few outliers that mostly occurred in the earliest trials for each input size. I re-ran the trials until I obtained data that was mostly uniform.

Table

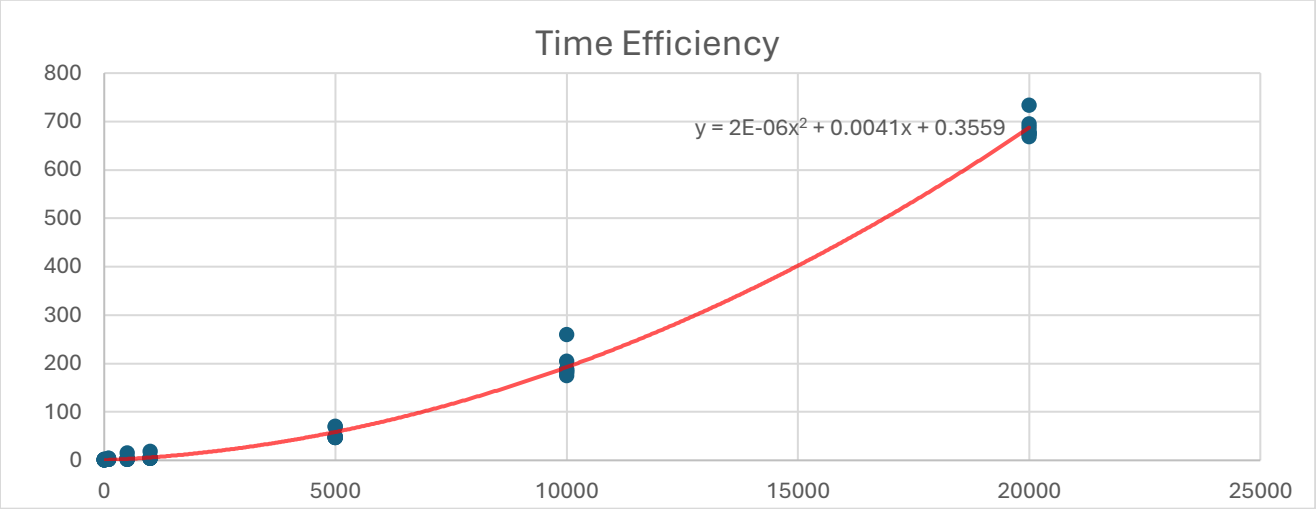
Time: milliseconds

Number of Queens (n)	10	100	1,000	5,000	10,000	20,000
Time for Trial 1	1	3	5	68	204	733
Time for Trial 2	<0.1	3	6	47	174	694
Time for Trial 3	1	1	4	70	259	674
Time for Trial 4	<0.1	1	4	49	182	678
Time for Trial 5	<0.1	1	4	47	181	687
Time for Trial 6	<0.1	1	4	47	188	668
Time for Trial 7	<0.1	1	4	46	186	674
Average	0.28	1.57	4.43	53.43	196.29	686.86

Graph

X-axis: input size

Y-axis: time (ms)



Analysis Comparison

Using the conclusion regarding the most-common case from the Theoretical Analysis in combination with the data and graph from the Empirical Analysis, the most-common case appears to fall around the $\Theta(n^2)$ efficiency class. This falls between both $O(n^3)$ and $\Omega(n)$ while also directly corresponding to the trend illustrated by the quadratic regression line for the data in the graph. Ultimately, both forms of analysis provide similar insight into the efficiency of the algorithm.