# NQueen Project Report

Dylan Chung

CS 4200.01 Spring 2020

Dominick Atanasio

March 07, 2020

**Program Description:**

The following program solves the NQueen Pair problem using either a genetic algorithm or simulated annealing and returns a solution, or in the case that it is unable to calculate the solution within the given constraints, it returns the best board it did find. It also outputs the time spent to find the solution and what generation it was on when it did.

**My Approach:**

To implement this program, I used a total of 5 classes; Main, NQueenGenetic, NQueenSimulatedAnnealing, Node, and QueenBoard.

Main holds the main function which runs the program. QueenBoard is a class which holds a single dimensional integer array that represents the n x n board with n queens. It also holds the fitness value and number of queens, and other various functions. QueenBoard is to be used with Node and NQueenGenetic.

NQueenGenetic uses an arraylist of QueenBoard objects. This allows for calling collection.sort() on the arrayList to sort by fitness value, so that we can select from the cream of the crop in each generation. It is based on the Genetic Algorithm. NQueenSimulatedAnnealing uses objects of class Node, which hold objects of class QueenBoard. It is based on the Simulated Annealing Algorithm.

**Shortcuts/Improvements Made:**

During the genetic algorithm, I gave the children a base 20% chance of mutating naturally, but certain conditions would force mutation automatically. For example, in the case where parent X and parent Y are exactly the same, I allowed the parents to breed, but I forced the resulting child to mutate. This would solve the problem of instances where they spent a long time working on the same boards until a mutation naturally occurred. I also saved the previous best board from the last generation, and in the next generation, if a child generated was exactly the same as the saved one from the last generation, a

mutation was forced. Finally, one step further, if the last child generated within the same generation was the same as the next child generated in the same generation, I forced mutation.

**Difficulties:**

Simulated Annealing cost me the better part of the day despite Genetic only taking me a few hours, but not because I didn't understand how the algorithm worked. Simply put, when I was generating successor boards, in which the program randomly moves a single queen to create a new board until it gets accepted, I was not resetting the board properly after every move. What ended up happening was, I had the foresight to think of resetting the board by copying the array into a temporary array, but did it improperly. I altered the original array by moving a queen, then set the board back to the temporary array, but did not realize the temporary array got changed when I altered the original because of how I had it set up. I totally forgot that arrays passed the reference of the values within them when you do OriginalArrays = TempArray, not the values. Took me too long to figure out what was wrong.

**Data:**

A 25N Queen Solution (Annealing) at 400 Instances with Initial Temp: 1000 and CoolDown Rate at 0.95 with a generational limit of 500 generations.
Avg Solved : 57.99999999999999%
Avg Generations : 407.3725 generations
AvgRunTime : 0.0038299954 seconds

A 25N Queen Solution (Genetic) at 400 Instances with a generational limit of 500 generations and Population of 100
Avg Solved : 96.5%
Avg Generations : 156.145 generations
AvgRunTime : 0.022262508 seconds

**Analysis:**

The data above is a compiled average of statistics from 400 instances of both algorithms. From the data, it seems that Annealing is the faster of the two. However, on average annealing is unable to calculate the right board in time before the generational limit (500 generations) about 43% of the time, whereas genetic only fails about 3.5% of the time and takes a lot less generations.

Interestingly though, despite taking less generations to solve, the average run time of the Genetic algorithm runs about 20 ms higher than annealing. I think this simply comes down to the fact that for each generation, genetic MUST generate the entire population size again before selecting two suitable parents. In Annealing, as soon as a board is generated, if the algorithm wants to accept it, then it gets accepted immediately. This can mean, the very first board annealing generates is accepted, and it's done with that generation, or in a very bad case, it could have very bad luck, and generate a ton of boards before it comes across one that is accepted. However, as shown by the stats, it seems is much faster on average to just generate one, check immediately, accept or reject, than to generate an entire population each time.

As for why Genetic has a higher success rate, I believe that is because of the constraints I put on the mutations (refer to Shortcuts/Improvements made section). I found situations where it would be convenient/smart to force a mutation so that we don't have generations that are too similar to the previous generation over and over again, thereby speeding up the process and lessening the amount of generations needed. With that, it is less likely to hit the generational limit I set for the purpose of this analysis. Annealing hits the limit more often perhaps due to my temperatures and cool down rates allowing the algorithm to be more lenient than optimal, accepting too many bad moves.