Charlson So
Project 2

General Architecture - Board

For the implementation of these algorithms, the fitness score was calculated using the number of non-attacking queens. The most optimal solution would result in a fitness of 25, indicating that the algorithm had successfully solved the n-queens problem. The initial queens are set using a method called set_queens_on_board() and would random generate an array representing the columns in a chess board. The values in the array would represent the row. Therefore, an array like [0,2,1] would be a 3x3 chess board with a queen in the following configurations:

first row, first column
third row, second column
second row, third column

A method called get_num_queens_attack would take in the (col, row) of a position and return the number of queens attacking the position given a board of queens. This is used in the function get_number_of_non_attacking_queens that would give us the fitness of board. Initially, I had created a different fitness function but opted to use the number of non attacking queens as my fitness function. The function is_valid_solution would tell us whether or not the board was a solution to the n queens problem.

change_state is a function utilized to generate a new state for the simulated annealing problem. This function would randomly choose a random column and change one of the queen values to a random value. This would generate a new state for the simulated annealing algorithm.

Simulated Annealing

The simulated annealing resulted in the most optimal solutions and had a 100% success rate. Although the algorithm should have resulted in solutions close to the optimal, tinkering with the threshold, decay_rate, and temperature resulted in solutions for the n-queens problem. In order to get more iteractions, I added more values to the decay rate so that the temperature would take longer to get to the threshold (e.g. 0.99 -> 0.999). The threshold can also be decreased for more iterations and by increasing the run time of the algorithm, optimal solutions can be generated. As a result, the simulated annealing algorithm would have an average run time of 7 seconds. The success of the algorithm is due in part to generating a maximum of random states.

This algorithm was not at all difficult to implement but an adjustment to the parameters such as temperature, threshold, and decay_rate was necessary to generate an optimal solution. Also, since my cost function would result in a negative value, the probability function needed adjustment to work correctly. Rather than using the textbook $p = e^{-c/t}$, the function was adjusted to $p = e^{c/t}$. I was rather surprised at the quality of my results.

Genetic Algorithm

This implementation of the genetic algorithm was successful. Given enough time, the algorithm will always result in a solution. I used a fitness function that equaled the number of queens that were non-attacking. The algorithm is able to successfully find a solution every time. On average, it took 328.83766 seconds to complete ans iteration of the algorithm. Although in some test runs, the algorithm could take longer than 30 minutes. The algorithm is able to find a solution every time because new generations are constantly created until a final solution is found.

In my implementation, I generate 700 random boards and then select 10% of the population with the highest fitness score. I then do a cross over where I choose 2 random boards and cross over at a random point. The max mutation rate of the algorithm was set at 50%. This means that up to 50% of any column would randomly move a queen. Although the algorithm is pretty slow at finding a solution, it shows that using the genetic algorithm will result in a solution. Some ways that I can think of to improve the performance of the algorithm is to create a fitness function that would better describe the performance of the solution in a more specific way. As the algorithm approaches a more optimal solution, the fitness of the boards generated cannot be differentiated and therefore will generate solutions that will not be moving in a direction closer to the solution. The algorithm would be more intelligent by having a more precise fitness function. Tinkering with the selection and mutation rate can optimize the speed of the algorithm. Although the specifics of the optimal rates are unknown, it is estimated that a population with low rate of selection will result in crossovers of very similar solutions while a high rate of selection will result in high variance of solutions with a low fitness score. A high mutation rate is similar. A high mutation rate will generate solutions with a low fitness score while a low mutation rate will not generate a variety of solutions.