SOLVING THE N QUEENS PROBLEM USING GENETIC ALGORITHMS

Computational Intelligence for Optimization

2023/2024

https://github.com/zepedrofcf/CIFO_nQueens

Group Name – Queen

20230525 Alícia Pinho Santos

20230522 Jaime Simões

20230983 José Pedro Cruz Fernandes

20222125 Nuno Sousa

Contents

Problem

The challenge of the **N Queens Problem** is placing N queens on a N-by-N chessboard, so that no two queens threaten each other.

This means that there cannot be two queens in the same row, column or diagonal. When n=1, there is a single obvious solution and for n equaling 2 or 3 there is no possible solution. However, for any integer n bigger than 3, there is at least one possible combination of positions that meets the criteria. This may seem a simple challenge, but as the n grows, the number of possible combinations for all the positions increases exponentially, requiring the optimization process to be efficient to solve the problem. Therefore, when the algorithm finds one of the solutions, the problem is considered as solved.

The developed algorithm begins by generating an initial population of possible random solutions. Then applies different selection, crossover and mutation functions over the population for a specified number of iterations (generations) until a solution is found (or the maximum number of generations is reached). The solutions coming from different implementations of these functions will then be compared using statistical tests to find the "best" combination for this specific problem. We chose to develop everything from scratch, instead of using the given library, so that a full understanding of the implementation would help us envision potential improvements. In this report, we will explain the implementation structure, covering all the different functions and parts of the code, and analyze the statistical tests and results.

Representation

The first thing that had to decide was how the problem would be represented in code (what would be the format of individuals, etc.) ending up choosing not to represent the board. Since we only need the work on the positions of the queen pieces, we decided that each board would only be n tuples of 2 integers. Each tuple contains the coordinates of a position, and the list of n tuple represents all n positions. This brought forward at least three things that should be considered:

- The order within an individual is irrelevant;
- Through mutation and crossover functions we need to verify that the resulting individual had no duplicated positions, and had to have size n;
- This structure led us to work with two random factors: x and y coordinates. Which is important because an alternative approach could have been to represent an individual as a simple list of integers, where the index corresponds to the other coordinate. That would have made things significantly easier.

Initialization and Structure

The algorithm starts by initializing a population of random individuals, each representing a set of n positions of random values for x and y coordinates, up to n. The size of this population is specified as a parameter beforehand and remains constant over the generations. Sequentially, fitness is evaluated for every individual. As we will see later, the fitness function is designed to approach this challenge as minimization problem. For this

reason, the initial fitness value is initially set as absurdly high value, ensuring that any first random position has a better score.

After the initial population is set, it is substituted according to the application of the chosen selection function on the original population.

Afterwards, if the option elitism is enabled, the specified percentage of the individuals with the lowest fitness are saved in a separate list which will become in the future the next generation's population.

The next step is to apply the crossover with the chosen crossover function to individuals chosen by the selection function and substitute the individuals in the process.

Lastly, to fill the rest of that "next population" list (where the elites were saved), mutation is applied to the current population (with selection to choose the "victims") and added to that final one until it is filled. The generation counts increments by one and the cycle repeats (except that the starting population is the one produced by the last generation and not a random one) until any individual has a fitness of zero. Throughout this process, there are several checks and controls to see if a solution has been found and several moments to update fitness values.

Fitness

Fitness defines the quality of an individual. In our case, it is defined as the number of conflicts between queens present in that individual (positions on a board), so a lower number means it's a better solution, turning it into a minimization problem for this value. If the fitness reaches 0, it means there are no conflicts and therefore the algorithm has reached a solution. There was really no other meaningful way to evaluate fitness differently, and so this was the only fitness function design.

Selection Functions

After evaluating the fitness, individuals are selected to form a new population. Selection is based on fitness, meaning that individuals with better solutions are more likely to be chosen. The algorithm implemented supports two selection methods:

- **Tournament Selection:** Two individuals are randomly chosen from the entire population and the one with the lower fitness is kept.
- **Roulette Wheel Selection:** Individuals are selected with a probably related to their fitness. So lower fitness values means that individual has a higher chance of being selected, by the same proportion of the fitness in relation to the fitness of the other individuals.

These selection functions are used in three moments: firstly, the whole population is selected on and substituted, which means that from the beginning of a generation there is a tendency so that the worst individuals are disregarded and the best individuals are kept; secondly, the selection function determines which pairs of individuals the crossover function will be applied to; lastly, it is used to select the individuals that will undergo a mutation.

Crossover Functions

The individuals selected are paired and produce offsprings through the crossover method, which combines parts from both parents to create new individuals for the population. Implementing the crossover method was particularly challenging, as order in an individual is irrelevant. The developed crossovers are:

- **crossHalf:** Takes the sorted positions of both parents and alternately distributes them between the offsprings. The process distributes the "genes" from both parents but if the parents have similar positions the offspring will closely resemble the parents.
- **crossSinglePoint:** Randomly selects a point and then the genes from the beginning to this point inherit from one parent with the remaining coming from the other. For the other offspring the process is symmetrical.
- **crossCycle:** Starting from the first position, it follows a cycle alternating between the parents by means of the output of each instance until it returns to the starting position. Then it fills the remaining positions with the parent not used as reference for that specific run.
- **crossGeometricSemantic:** This method uses random weights to combine the coordinates from both parents. Because it can produce non integers, the values are rounded to the nearest integer for the coordinate value.

Mutation Functions

Taking again the selection function to choose individuals to apply mutation on, the variability of the population is increased. Two possibilities exist for this step:

- **Individual for Random:** regenerates that individual randomly.
- **Position with conflict for random:** Finds a position in a conflict with another and replaces the position of that queen by a random new position.
- **Shift Coordinate on Position with Conflict:** Finds a position with conflict and shifts just one of the coordinates (either row or column, chosen randomly).

Elitism

Elitism can either be enabled or not (True and False). In the case where it is enabled, for every generation the algorithm preserves the best performing percentage of individuals (which we kept at 10%) and carries them to the next generation. This ensures that these top performers are not lost due to the crossover or mutation phases. For the population to maintain its size, the elites are saved in a list, and that list is the one that will be used as the next generation: the results from crossover and mutation simply are added to that list until it reaches the population size. Alongside the different options for the functions explained previously, elitism is another parameter that will be optimized when running for solutions.

Looking for the best model

Since there are many parameters and function options for several steps of the algorithm, we had to search independently in different sections of the possible search space. This may not be considered the optimal or most accurate way to obtain the absolute

best model but given the computational complexity and time we had for this project, we found it to be a good approximation.

Our analysis primarily relies on graphic representation. Each dot on the graph corresponds to the average time of 30 runs for a specific combination of a board size. To make the graphs possible to read we will join the points of different board sizes with lines, however this method does not provide an entirely accurate representation of the problem.

First, we searched for the best combination of functions for crossover, mutation, and selection (using population size of 150). Then we optimized the size of the initial population and evaluated whether using elitism was beneficial. Once we determined the "best model" we ran it more thoroughly to obtain the final results. Finally, we analyze how the randomly set initial positions impacted these results.

Finding the best combination of functions

Due to the numerous types of functions for "selection", "mutation" and "crossover", and the fact that some of them take a long time to run, to compare them all we started by running a smaller (n) so that we could have an idea of which functions perform better or worse. Then for the "final" comparison we ran only the best combinations to a bigger board size. The most accurate way would be to just directly compare all possible combinations, but as this is not computationally feasible, we decided to take this approach.

When seeing the outputs of the first runs it was clear the crossover function "crossCycle" was performing poorly against the others, so it was removed from this analysis. The same was applied to the "mutate for random" mutation. Figure 1 displays the results up to n=6. It is notable that the "Tournament Selection" appears to outperform "Roulette Wheel Selection" with lower lines, meaning that it takes less time to solve the problem. Therefore, we chose to use "Tournament Selection" exclusively going forward. It was also possible to identify that for crossovers, "crossGeometricSemantic" had the worst performance for both selection functions, leading us to keep studying only the other two crossover functions.

With this information in hand, we increased the size of the board to get a better estimate and determine what was the best combination. Figure 2 illustrates the comparison between the remaining combinations. The best performance is obtained for "Tournament Selection", "Position with conflict for random" and "crossSinglePoint". We then proceeded to do tests with other parameters on this combination.

Optimizing population size and elitism

Following the same logic as before we continued to run the best combination for different population sizes and elitism options to compare the results. When elitism is "True", the fraction of the best population kept in order of best fitness is always 10%. Upon examining the results of Figure 3, the best model changes depending on board size. For smaller sizes, the solving speed is faster, but it starts to increase significantly for higher sizes. Therefore, it is preferable to choose a model that works better with higher board sizes, as that is what will save the most amount of time in the long run. With that, population size of 100 and elitism being enabled seem to be the better option. Now we can proceed to run the "best" model.

Results and associated fluctuations

On Figure 4 we can see that the growth in time is approximately exponential as would be expected until n=17. Now, we can do analyze how the initial positions, which are set randomly, affect the time it takes to solve the positions. For that we maintained all the parameters constant and obtained the distribution of the results for each run via boxplot.

Using the parameters of the best model, in Figure 5 we follow this logic for different board sizes. The horizontal line of 0 indicates the average for each instance. There are differences higher than 50%, which means the initial positions have a big impact on the results. The problem is seen across different board sizes. This might indicate that we should have used more than 30 runs for each point for the graphs above to get more consistent results and that the performance of the algorithm is dependent on a random factor, at least for these values of (n). This also helps explain the fact that in Figure 4 from 16 to 17 there is a decrease in time, we also see an increase probably higher than expected from 15 to 16.

Addendum

Comparison of Genetic Algorithm with other Algorithms

As a final part of our analysis, we conducted a comparison of the genetic algorithm's performance against three other algorithms sometimes used to solve the N-Queens problem: brute force, backtracking, and simulated annealing.

- **Brute Force**: This approach systematically enumerates all possible configurations for the queens on the board and checks each one to see if it's a solution.
- **Backtracking**: This algorithm builds up to a solution incrementally, abandoning a solution as soon as it determines that the queen placed at the nth column does not lead to a solution. We included a version that stops after one solution and one that finds all solutions.
- **Simulated Annealing:** A probabilistic technique for approximating the global optimum of a given function. It attempts to avoid being trapped in local optima by allowing less optimal moves but gradually reduces the probability of such moves as it explores more solutions.

Brute force is perhaps the most inefficient possible algorithm, besides random search, and without a logarithmic scale would start to make it impossible to detect any distinctions after ~n=12. Backtracking follows behind, although the difference remains large between the two. Figure 6

Simulated annealing increases in a much more progressive manner (although its behavior, which is highly dependent on the parameters, could perhaps be modified if the parameters were explored further) contrasting with the genetic algorithm which behaves in a more “stochastic” manner. Figure 7

This comparison serves mostly as a sanity check method to validate the optimization of our own genetic algorithm implementation, by comparing it with other popular algorithms which are well benchmarked.

Annex

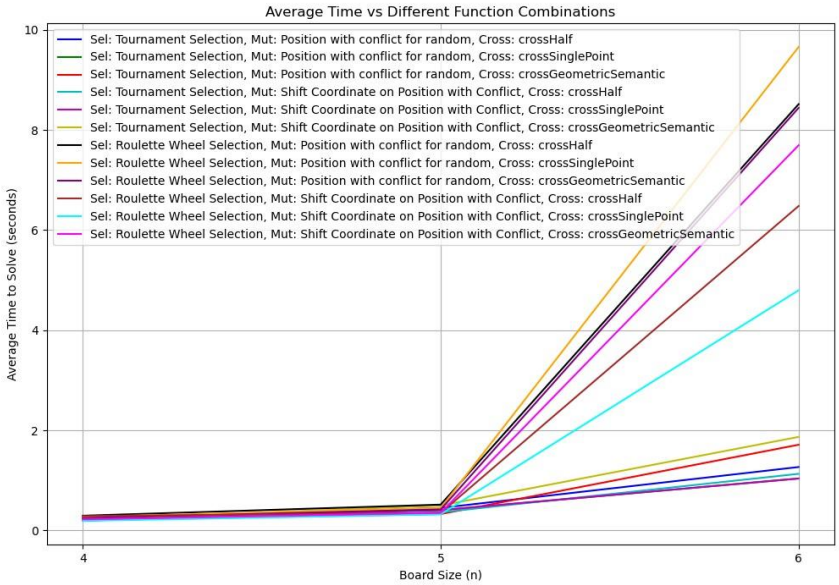

Figure 1: First comparison to get the best combination of functions

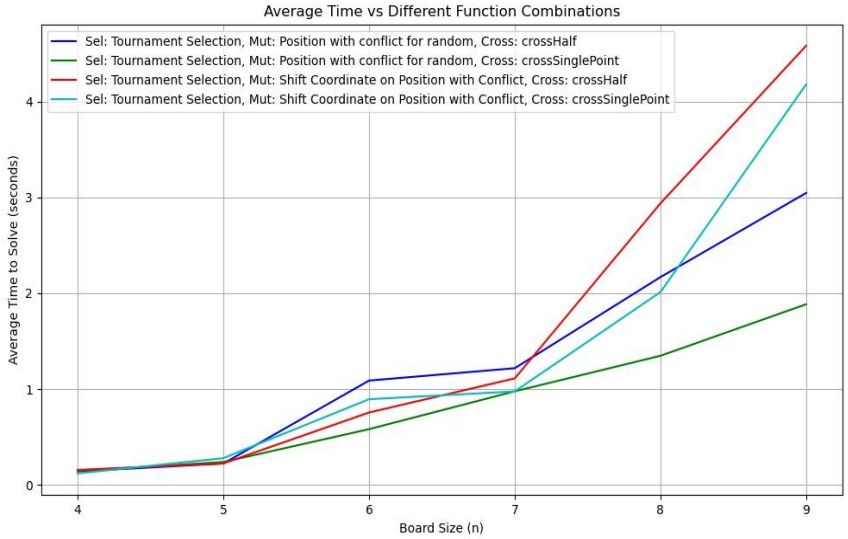

Figure 2: Finding the best combination of functions

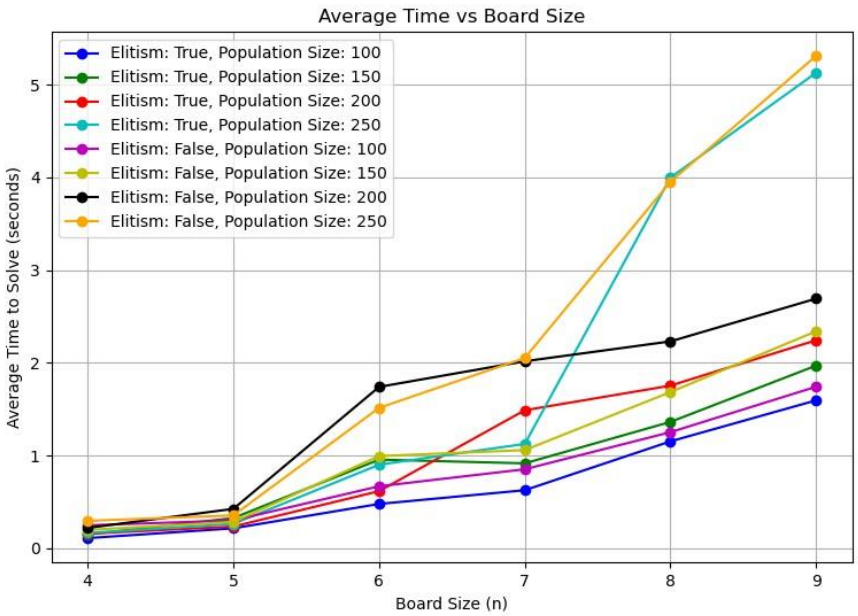

Figure 3: Different population sizes and elitism option

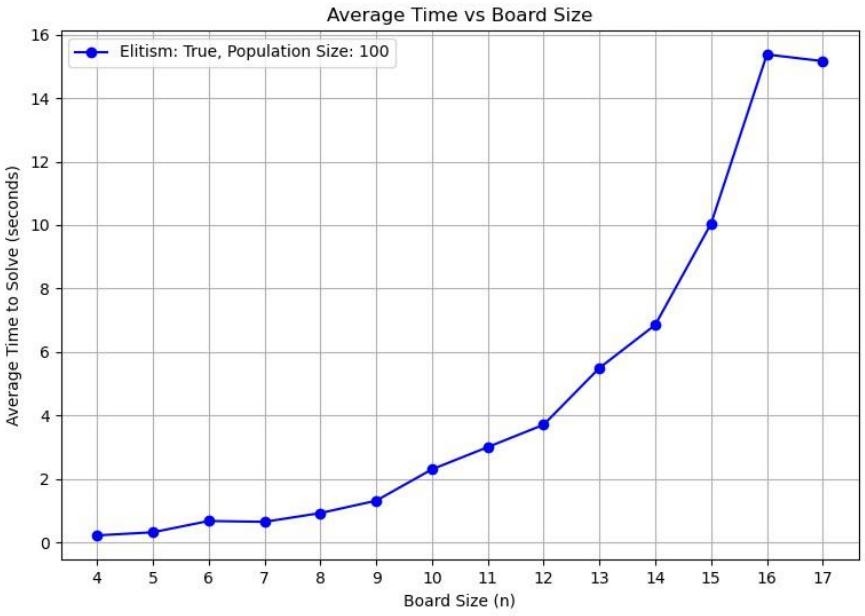

Figure 4: Results for the best model

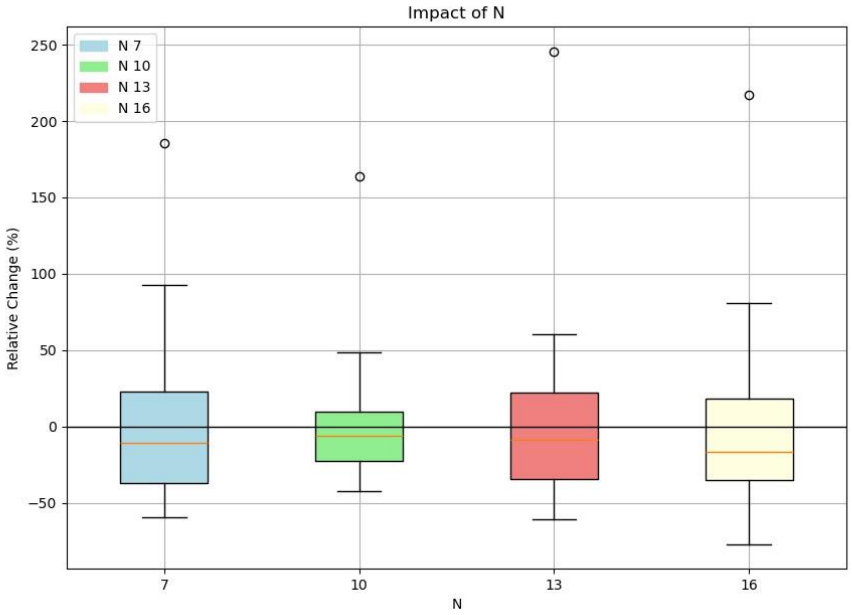

Figure 5: Variation of results on different runs

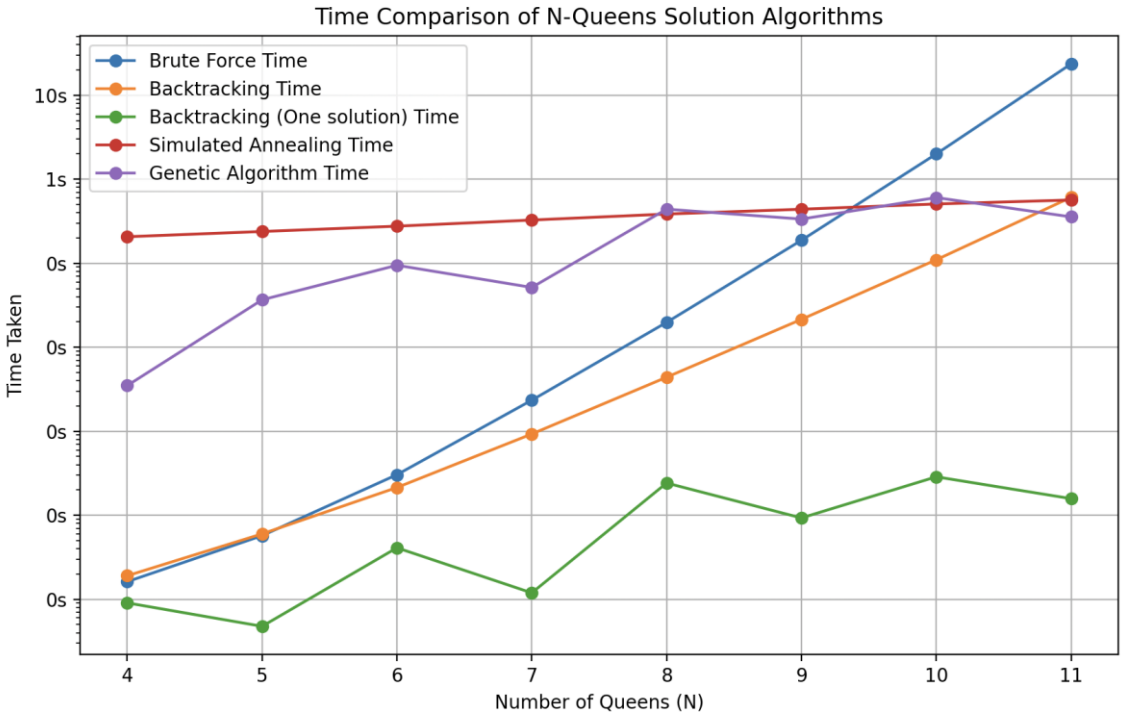

Figure 6: Comparison of all other algorithms (log scale)

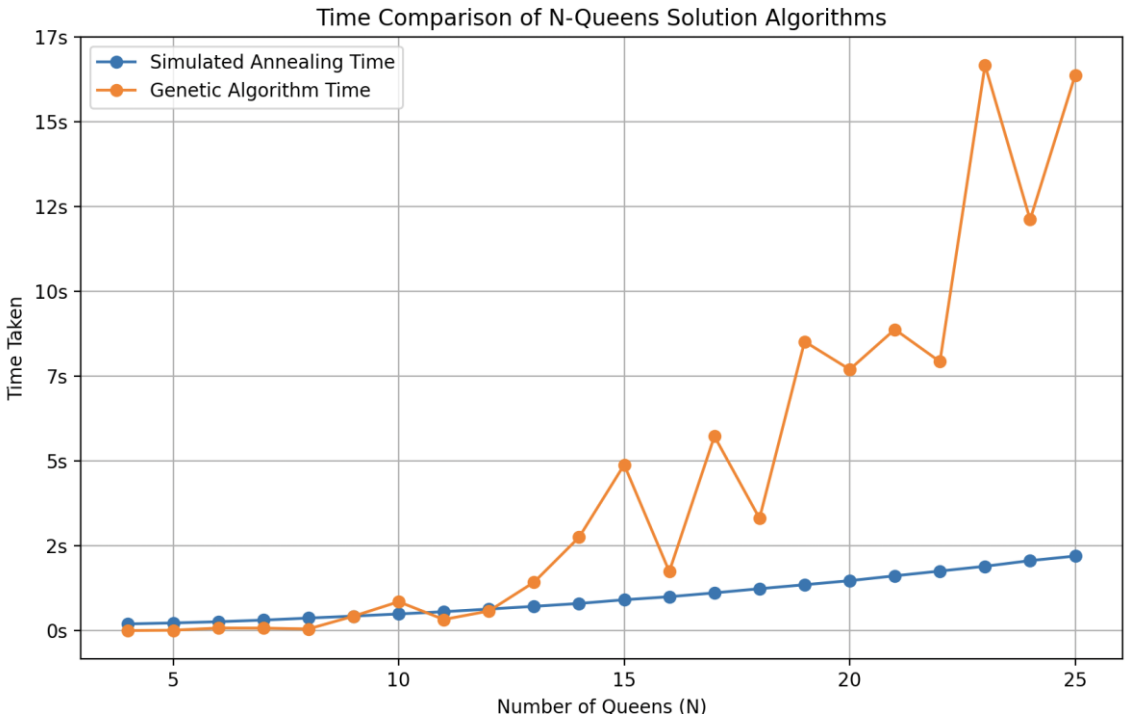

Figure 7: Simulated annealing and genetic algorithm

Division of labour

Alícia Pinho Santos: Research, Implementation, Report

Jaime Simões: Research, Implementation, Statistical Analysis, Report

José Pedro Cruz Fernandes: Research, Implementation, Report

Nuno Sousa: Research, Further comparison research, implementation and analysis, Report