**Algorithm:**

**Problem:**

The concept of the N Queens problem is having a chess board of size N\*N with N of queens in a combination of positions such that two queens cannot “check” each other, so there can not be two queens present in the same row, column or diagonal. With the increase of N, the number of possible combinations for all the positions increases exponentially, requiring the optimization process be efficient to solve the problem. For any n bigger than 3, there is one or more combinations of positions that meet the criteria. For an n equal to 1 there is a single obvious solution and for n equalling 2 or 3 there is no solution. When the algorithm finds one of the solutions, the problem is considered as solved.

The developed algorithm generates an initial population of possible random solutions, and through different possibilities for the selection, crossover and mutation functions will iterate until it finds a solution. The solutions coming from different implementations of these functions will then be compared through statistical tests in order to find what could be the “best” combination of these functions for this specific problem. We decided to construct a different library of code to the one developed in the practical classes of this course. We will go through this code following the chronological order it runs when solving the problem.

**Initialization:**

The algorithm begins by initializing a population of random potential solutions. The size of this population is a parameter that will be optimized but it remains constant over the generations. The positions of the queens will be represented as a (x,y) coordinate system where x is the row and y is the columns on the board of size N. For the initial population (generation 0), these coordinates are set randomly but making sure that there are no groups of queens either on the same row or on the same column, remaining inside the bounds of the board chosen. Making an initialization with random parameters but also some “rules” ensures that the initial population is already a bit oriented for the solution but also has the diversity needed for a genetic algorithm. As we will see later, the fitness function is made in a way that it becomes a minimization problem. For this reason, the initial fitness value is set very high.

**Fitness:**

The fitness value is what defines the quality an individual (a set of positions). It is defined as the number of conflicts present in that 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 algorithm has reached a solution.

**Selection Functions:**

After evaluating fitness, individuals are selected to form a new population. Selection is based on fitness, with better solutions having a higher chance of being chosen. The algorithm 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.

This selection function will be used in there 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 is used to decided which pairs of individuals are going to have a crossover applied on; lastly, the selection function will be used to select the individuals which will undergo a mutation.

**Crossover Function:**

The individuals selected and paired and produce offspring through the crossover method, where it combines parts from both parents to create new individuals for the population. The crossovers developed are:

* **crossHalf**: takes the sorted positions of both parents and distributes them alternately between the offspings. It distributes the “genes” from both parents but if the parents have similar positions the offspring will be more equal to the parents.
* **crossSinglePoint**: selects randomly a point and then the genes from the start to this point come from a parent with the rest coming from the other parent. For the other offspring is the 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 Function:**

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

* **Position with conflict for random**: picks a position where there is a conflict and replaces the position of the queen by a random new position.
* **Shift Coordinate on Position with Conflict**: also targets the positions where there are conflicts. But this time it shifts just one of the coordinates (either row or column). And this times it applies this to all present conflicts.

**Elitism:**

There is the option to have elitism on or off. In the case where it is on, for every generation it saves the best 10% of individuals with the best fitness values and copies in the next generation, so that these are not lost in the crossover / mutation process. For the population to maintain the same size, it only creates new elements until that threshold is reached. So, it will create less individuals then if elitism is set to False. Alongside the different options for the functions explained above, this is also a parameter that will be optimized when running for solutions.

**Looking for the best model:**

Because there are many parameters and function options for several steps of the algorithm, we have to search independently in different sections of the possible search space. Now this is not the optimal or most accurate way to obtain the absolute best model but given the computational complexity and time we have for this project, we find this to be a good approximation. Each dot corresponding to each board size is the average time of 30 runs for that specific combination. To make the graphs possible to read we will join the points of different board sizes with lines, but this of course is not an accurate representation of the problem.

First, we look for the best combination of functions for crossover, mutation and selection (using population size of 200). Then we optimize for the size of the initial population and if using elitism is beneficial or not. After this we have the “best model” and will run it more thoroughly to get the final results. We then analyse how the initial positions that are set randomly impact these results.

**Finding the best combination of functions:**

Because there are many different types of functions for “selection”, “mutation” and “crossover”, and some of them take a long time to run. In order to compare them all we will run by sections so that we progressively see which of these combinations gives better results, also just looking at low values of n (size of board). As the best combination becomes more apparent, we will look at higher values for this parameter. 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. In figure (a) we have all the selection and mutation functions but just the first two crossover functions. It is notable that the “tournament selection” selection has the lines above “Roulette Wheel Selection”, meaning it is performing worse. We only use Roulette from now onwards. For mutation, “Individual For Random” is performing the worse for both selection functions, so we will only keep the other two. Now we take this information and add the other crossover functions that weren’t used on the first run. Looking at figure (b), for the same selection and mutation, the crossovers “crossHalf” and “crossSinglePoint” have better performances than “crossCycle” and “crossGeometricSemantic”, so will only keep the first two. Now that we have less combinations we will run for higher board sizes to make a better assessment of what the best combination of functions is. Looking at figure (c) we will take the lowest line as the best combination, assuming that it would maintains bellow the other ones for higher board sizes. So that is “Roulette Wheel Selection” for selection function, “Shift Coordinate on Position with Conflict” for mutation function and “crossHalf” for crossover function.

**Optimizing population size and elitism:**

We follow the same logic as before and run the best combination for different population sizes and elitism option to compare the results. When elitism is “True”, the fraction of the best population kept in order of best fitness is always 10%. Looking at the results (figure (d)), the best model changes depending on board size. Because for low sizes the solving speeds are fast but start to increase significantly for higher sizes, it is preferable to choose a model that is the best one for 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 that we have all the optimized parameters, we can run the “best” model.

**Results and associated fluctuations:**

Running until n=9, we can see that the growth in time is exponential as would be expected (figure (e)). We can now do some analysis on how the initial positions, that are set randomly, affect the time it takes to solve the positions. For that we maintain all the parameters constant and obtain the distribution of the results for each run via boxplot. Using the parameters of the best model, in figure (f) we follow this logic for different board sizes. The horizontal line of 0 indicates the average. There are differences higher than a factor of 2, which is very significant. The problem is seen across different sizes equally for small values. 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 very dependent on a random factor, which is not a great sign.

**Annex:**

Uma imagem com texto, captura de ecrã, file, Paralelo

Descrição gerada automaticamente

Figure (a): First comparison to get the best combination of functions.

Uma imagem com texto, captura de ecrã, file, Paralelo

Descrição gerada automaticamente

Figure (b): Second comparison to get the best combination of functions.

Uma imagem com texto, file, diagrama, captura de ecrã

Descrição gerada automaticamente

Figure (c): Getting the set of functions with best performance for low (n).

Uma imagem com texto, captura de ecrã, diagrama, file

Descrição gerada automaticamente

Figure (d): Different population sizes and elitism option.

Uma imagem com texto, file, diagrama, Gráfico

Descrição gerada automaticamente

Figure (e): Results for the best model.

Uma imagem com diagrama, texto, Esquema, file

Descrição gerada automaticamente

Figure (f): Variation of results on different runs.