Computational Intelligence for Optimization

MASTER DEGREE PROGRAM IN DATA SCIENCE AND ADVANCED ANALYTICS

The N-queens problem

Group 16

Adriana Monteiro, nº20220604

Gonçalo Aires, nº 20220645

Pedro Ferreira, nº 20220589

Quintino Fernandes, nº 20220634

Link to Github: https://github.com/adriana-monteiro/project-cifo

NOVA Information Management School
Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Division of Labour: Adriana 27%, Gonçalo 20%, Pedro 27%, Quintino 26%

1- Introduction and Problem definition

N-queens has the goal of setting n queens in an $n \times n$ chessboard in a way where they do not kill each other. Our problem is actually a variation on the way n-queens is usually approached, because it is usually solved knowing that the solution has n queens. We thought it would be more challenging not to "tell" the algorithm that the perfect solution would have n queens, as it would increase the complexity if the problem. The standard n-queens problem has $C_n^{n^2}$ possible settings of queens on the board, but our specific problem had 2^{n^2} possibilities, which are a lot more. This means there is an increase in search space size compared to the traditional problem. For example, if we have a very small 4x4 chessboard, the original n-queens would have 1820 possible combinations. In our problem, however, the same chessboard could distribute the queens in 65536 different ways!

N-queens' solutions increase in complexity with the size of the board, but we wanted to make a general solution, so one could use genetic algorithms – and particularly the one we ended up choosing – to solve the problem for an arbitrary n. The optimal solution for this problem would have zero dead queens, while having n queens set around the board.

2- Representation

We wanted to keep our representation as simple as possible so it could work well with the operators we developed in class with some adjustments, so we decided on a binary number representation. Zero means there is no queen in a square, while 1 means that there is. This representation means that the board's lines and columns are not physically separated, as we represented the binary number as a list of zeros and ones, with length $n \times n$. To illustrate how our representation, let's imagine a 4x4 board that has a queen on the first square and on the last (upper left and lower right corners of the board). Its representation would be the following: [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1].

By representing our chessboard with a binary number, we could use most operators in our problem, while having to adapt some of them. This representation allows for an easy read of what is the solution, while also always having the same length of $n \times n$, which is crucial in solving problems with genetic algorithms.

3- Fitness Definition

As we said, our approach was for the algorithm to put queens on the board without them killing each other, but not knowing the number of queens it could or should set. Because of that, our problem had a whole different dimension of optimization, because not only did we want to minimize queens' deaths, but also wanted the number of queens to be maximal. This meant that we had two goals instead of one for our fitness: minimize deaths, maximize queens.

The first approach we thought of was of having a fitness function that returned a tuple, with the number of queens and deaths. But we figured that this would jeopardize the usage of most operators we were familiar with, so we added a layer of weighted sum of those two values for it to return just a number, in order to be able to do minimization or maximization to find the optimal solution. We ended up choosing minimization, which meant that our coefficients would have to be positive for the number of deaths, so we would get the smaller number of deaths possible, while the number of queens' coefficient had to be negative, so that the more the number of queens, the lower the fitness would be, and better the fitness.

But having less causalities was more important than having more queens on the board, so we used a coefficient with greater absolute value for the number of deaths, so this variable would be more important for fitness calculation. This entails that a solution with three queens and zero deaths was better than one with four queens and one death. We also wanted to penalize the choice of an individual with all queens dead, so, in that case, we multiplied the fitness by n, increasing the fitness by a lot if that happened. So, this fitness function had the following form:

$$\text{Fitness} = \begin{cases} 5 \times deaths \ - \ 2.5 \times queens, if \ deaths \neq queens \\ n \times (5 \times deaths \ - \ 2.5 \times queens), if \ deaths = queens \end{cases}$$

We chose the coefficients' values in a heuristic manner, with the requirement of settings like the one we just talked about to have less fitness for three queens and zero deaths (fitness = -7.5) than one with four queens and one death (fitness = -5).

Because of natural evolution in the project's plan, we ended up having a fairly simple fitness function, in the sense that we decided it would be better to have the individual itself have attributes like the number of deaths and queens, to make it easier to get those things without having to rely on the fitness function. Moreover, that meant we could use double tournament selection, which will be talked about later – where we needed a tuple representation of fitness but also a single number output for elitism or comparison between the used operator. In that sense, while not explicitly being a fitness function, these two attributes worked like one in the case of double tournament selection.

4- Operators and comparisons

We tried three different mutation, crossover and selection operators. For the most part, we wanted to try every combination possible, not only of the type of operator, but also the probabilities within them, but for the case of double and simple tournament, we had to compare tournament sizes beforehand, to only put the best one on the final combination search.

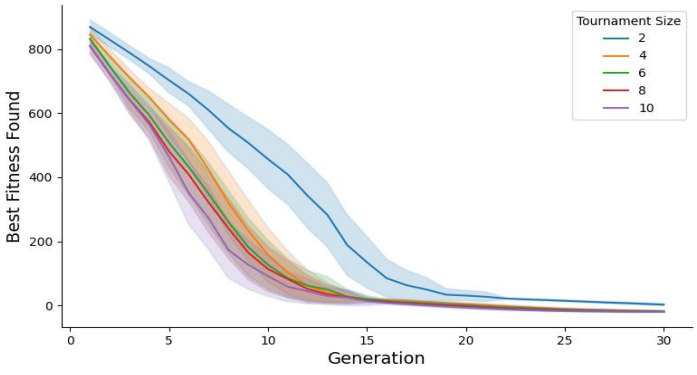
Before starting performance comparison, we first had to define how we could make an experiment that could be used for comparison between models. To do so, we created a function that allowed us to choose the number of runs we wanted to do with each combination, or each try, because we wanted to have a statistical significance in our comparisons, which meant that we'd have to try the same configuration thirty or more times because of the probabilistic nature of genetic algorithms. This function also allowed for a smoother experience of running each try and later put it inside the grid-like search for the best combinations, as it allowed us to also choose all the things we have to when evolving a population. What this means is that in practice, if we do 30 runs of a certain algorithm that combines different operators, in the end we will get a mean of the parameters (fitness, deaths and queens) over all runs, for each generation. This way we can visualize it later with statistically significant values. If we choose 30 runs, then the grid-like search will do 30 runs for each configuration that we want to try, so the comparison is possible between genetic algorithms with different mutation and crossover probabilities, mutation, crossover and selection operators.

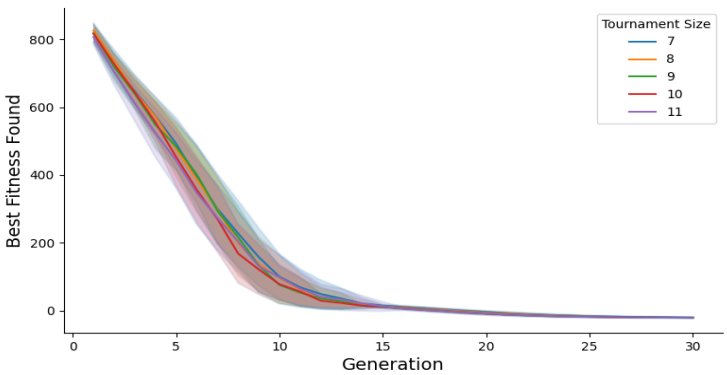
For our experiments, we chose n = 10, a population size of 200 and 30 generations each, as that was enough to achieve a fitness plateau. The number of runs depended on what we were testing – 30 runs for most of them except for tournament selection, elitism and to test the final solution, where we used 100. In the illustrative plots, the translucent colour represents the standard deviation around the mean.

4.1- Tournament size choice

4.1.1- Tournament selection

So, first of all we tried tournament selection with different sizes ceteris paribus (binary mutation with probability 0.2 and single point crossover with probability 0.9), so we could see what size of tournament suited our problem the best. First, we tried with the sizes [2,4,6,8,10] and with the plots and the values of the best average fitness per generation we came to the conclusion that the best tournament size would





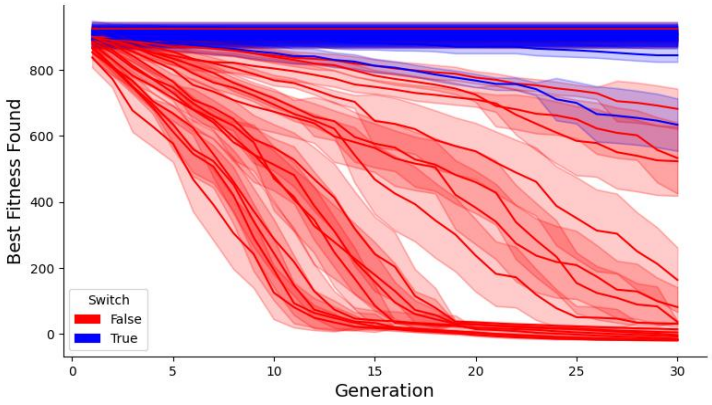
be above 8, since the difference between the values of fitness from the 6 and 8 tournament sizes were considerate, while also the convergence rate being faster.

So, we did a second try with sizes [7,8,9,10,11]. We found that 9 and 11 were the best tournament sizes in terms of fitness, with 9 having a slight margin in convergence speed.

4.1.2- Double Tournament

This algorithm is more complex than the normal tournament selection, since it introduces a second round where individuals are evaluated on another criterion. On this tournament we do not select the individuals based on fitness, but on the number of queens and their deaths. Double tournament has two rounds: in the first one, *deaths_t_size* determines the number of tournaments done; each of these will have *tournament_size* size and the criteria for selection is the least number of deaths. In the second one, we sample *queens_t_size* winners of the previous round to compete according to their number of queens – higher is better. It is usually implemented with switch, which determines the order of the criteria. In our case, we thought it made more sense to keep switch false, so the number of deaths was the first criterion.

We implemented the grid-like way to search for different combinations of parameters (or operators). Firstly, we wanted to test if switching the order of the tournament would get any meaningful results and, as we can see with the following plot, it didn't, as we expected, as it got higher fitness values and converging a lot slower or not at all.

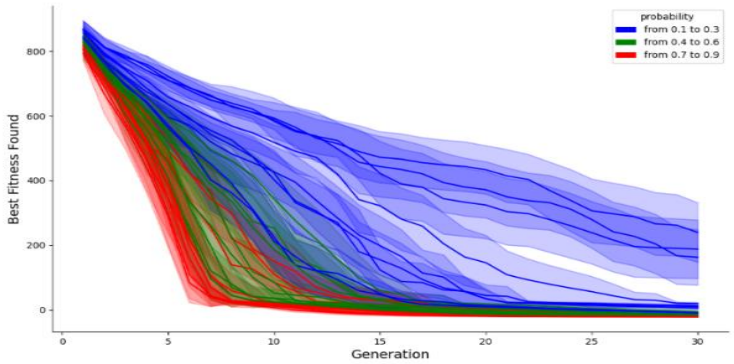


Secondly, we tested the sizes of the tournaments and found that the best initial tournament size was 10, then for the death's tournament best size it was 6 by a large margin. The best size for the second tournament that selects based on the number of queens, was 2, as highlighted in the table below.

Tournament_size	Deaths_t_size	Queens_t_size	Switch	Best Fitness Avg
10	6	2	False	-20.8333
10	8	2	False	-19.9167
10	10	2	False	-19.8333
8	6	2	False	-19.8333
10	4	2	False	-19.75

4.2- Crossover Probability

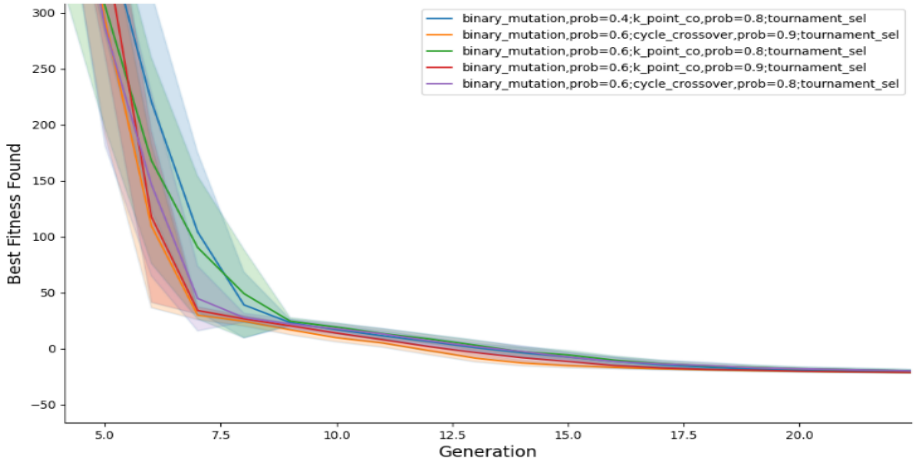
For crossovers, we wanted to know what probability was better. So, we found that with Cycle Crossover and K point Crossover with k=3, the best crossover probabilities were between [0.7,0.9]. As we can see in the plot, the red lines are the ones that converge the fastest in the majority of the runs. So, higher crossover probabilities seem to be better for our optimization problem.



4.3- Grid-like Search

Having found the best crossover probabilities and the tournaments' parameters, we had to test all different combinations of operators and probabilities with the grid-like search we developed, analogous to what we did before. We chose to do the search with the single tournament only because of computational limitations - we tested different selection methods afterwards. So, we tested the following operators and parameters:

- Mutation: Binary mutation, Swap mutation and Inversion Mutation (probabilities [0.2, 0.4, 0.6, 0.8, 0.9])
- Crossover: Single-point crossover, K-point crossover and Cycle crossover. (Prob= [0.7, 0.8, 0.9])
- Selection: Tournament selection (size = 9)



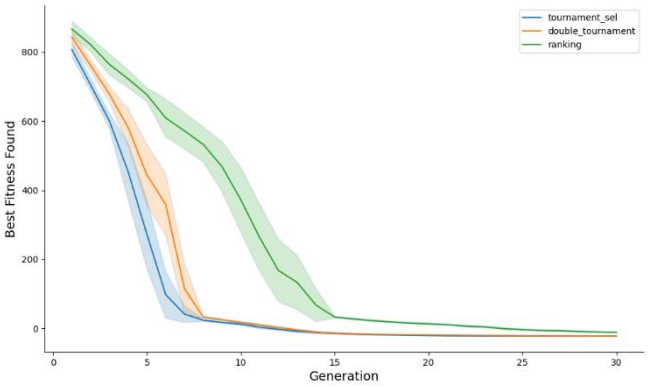
Mutation	Mut_prob	Crossover	Xo_prob	Selection	Best Fitness Avg
Binary mutation	0.4	K_point_co	0.8	Tournament_selection	**-22**
Binary mutation	0.6	Cycle_crossover	0.9	Tournament_selection	**-22**
Binary mutation	0.6	K_point_co	0.8	Tournament_selection	**-21,9167**
Binary mutation	0.6	K_point_co	0.9	Tournament_selection	**-21,9167**
Binary mutation	0.6	Cycle_crossover	0.8	Tournament_selection	**-21,75**

From the combinations of mutation and crossover methods and their probabilities with the best fitness average we ended up choosing Binary mutation with a probability of 0.6 and Cycle crossover with a probability of 0.9, because according to the plot above it seemed to have a higher convergence speed.

4.4- Selection

After finding the other best operators and their best parameters, we wanted to know what would be the best selection algorithm, so we did a grid-like search ceteris paribus and found that the Tournament selection would be more appropriate even though it has a slightly less good fitness it has a better convergence rate than the Double tournament. This is added to its better computational efficiency to make it a better choice.

Selection	Best Fitness Avg
Double_tournament	-22.1667
Tournament_selection	-22
Ranking	-11,333



5- Elitism

With the best combination of operators, we achieved previously, we tried to turn off elitism. Our expectations were that elitism would get a better result, but in the end the average best average fitness for all the runs was very close and convergence rate was very similar as well. In spite of that, elitism turned on seemed to achieve better results overall.

Elitism	Best Fitness Avg
True	-21,825
False	-21,725

6- Final Algorithm and Conclusions

We ended up with the following algorithm: tournament selection with size 9, binary mutation with probability 0.6 and cycle crossover with probability 0.9.

Because n-queens is a solved problem, we wanted to check if our final algorithm would be able to reach a solution (since there are many) for two different n values. We ran the algorithm 100 times for a 10x10 chessboard and found three unique solutions. For n=12 it only found one unique solution in 30 runs. Because for these values the search space is enormous (orders of magnitude of 10^{30} and 10^{43} respectively), we expect the algorithm to be more proficient in finding solutions for smaller values of n. Overall, our algorithm reached very decent results. If we adapted our problem to need maximization, then our implementation would work.

We could try to improve the project by implementing elitism based on non-dominated solutions, which by using it with double tournament would make the algorithm independent from a fitness function that depends on weights that are either heuristically chosen. Another improvement would be to try to optimize the weights of the fitness function themselves. It could help to try more operators and combinations, but more computational resources would be needed. It would also be great to try the algorithm and even optimize it for bigger chessboards, as the solutions are more complex and difficult to find by hand.

7- References

1 - Vanneschi, L.; Silva, S. (2023). Lectures on Intelligent Systems. Springer International Publishing. https://doi.org/10.1007/978-3-031-17922-8

2 - Luke, S., & Panait, L. (2002). Fighting bloat with nonparametric parsimony pressure. Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 2439. https://doi.org/10.1007/3-540-45712-7_40