A Genetic Algorithm for the 8-Queens Problem

Moran Gybels and Andreas Radke

December 10, 2017

1 Introduction

This report describes a solution to the n-queens problem. The 8-queens problem is an example of this more general problem. The problem is solved by using a genetic algorithm. Genetic algorithms are based upon the evolution theory, the natural selection. They rely on operators, inspired on biology, such as mutation and crossover.

The algorithm was implemented in Python 3.6+ with the aid of the Numpy library.

2 Problem Statement

The n-queens problem is the problem of trying to put n queens on a $n \times n$ chessboard in a way they are not attacking one another. The problem had to be solved using a genetic algorithm that was developed "ad-hoc".

3 Algorithm

The general overview of the genetic algorithm can be described as follows:

1. Generate random population and compute the fitness of each individual.
2. Compute next generation of population by:
   (a) (Optional) Copy a small amount of (fit) individuals from the current generation to the new one to guarantee fit individuals.
   (b) *Selection*: Choose two random individuals for mating from the population. The algorithm should tend to choose fitter individuals.
   (c) *Crossover*: Mix them with a specific probability via a *crossover* method to produce two children for the next generation.
   (d) *Mutation*: Let them mutate, i.e. change its values or *genotype*, with a low probability.
   (e) Insert children to next generation and repeat until the next generation has the same size as the one before.
3. Repeat 2 until optimal solution is found or a maximum number of iterations is reached.

4. Output fittest individual.

Our genetic algorithm starts with creating a random initial population, possible solutions to the problem. In a possible solution, called an *organism*, the n queens are represented in a list of length n containing values ranging from 0 to n-1. n will be the size of the chessboard, often called field size.

Each index represents a queen on the chessboard. The index of the list determines the row and the value of $list[i]$ corresponds to the column. This list is called *genotype* and its elements *genes*. It is obvious that there is only one queen per row. But with a uniqueness test and careful crossover and mutation operations we also made sure that there is also only one queen per column, i.e. every value can only occur once. One example for an *organism* is

$$genotype = [1, 4, 3, 7, 5, 6, 2, 0]$$

The algorithm computes the fitness of each organism. The fitness is calculated by counting the number of times queens can attack one another (only diagonally in our case) and subtracting that from the maximum amount of possible attacks:

$$\frac{n(n-1)}{2},$$

i.e. each queen is placed on the same diagonal ($genotype = [0, 1, 2, 3, 4, 5, 6, 7]$).

To calculate the next generation, the genetic algorithm goes through the above mentioned steps such as selection, crossover and mutation. The algorithm implements multiple methods with different parameters for the three stages.

The ideas behind the following methods found in the sections 3.1, 3.2 and 3.3 are mostly taken from [1].

3.1 Selection Methods

For selecting the parents of the next generation there are different methods implemented.

- *Roulette*: Compute the accumulated fitnesses of the current population and view it as one long contiguous line with each segment corresponding to one individual. Take a random point on this line. The better the fitness the longer the segment and therefore the probability to be chosen.
- *Truncation*: Truncate the current population by a specific percentage such that only the fittest $x\%$ are considered. Choose from them randomly. A low x value is exploitative.
- *Tournament*: Select a x tournament competitors (organisms) from the current population randomly. Choose then the fittest among the competitors. Higher values for x are more elitist (exploitatory), lower values are more exploratory.
- *Random*: One of the above is randomly selected.

3.2 Crossover Methods

The following crossovers are implemented:

- *Order Based*: Choose randomly several points in the first parent and fix the order of their appearance. Now look for their values in the second parent and rearrange them such that they match this order. The rest from the child's genotype is taken from the second parent.
- *Position Based*: Choose randomly several points in the first parent and fix the position of them. These positions and its values are applied to the child. The rest (i.e. the remaining indices or rows) is then filled up with the genes from the second parents.
- *Partially Matched*: Also called Partially mapped crossover or short *pmx*, because parents are mapped to each other. Choose randomly two crossover points and map the segment content from one organism to the other. The remaining information is exchanged through matching.
- *Random*: One of the above is randomly selected.

3.3 Mutation Methods

The following mutation methods are implemented:

- *Exchange*: Choose randomly two rows/genes and swap them.
- *Scramble*: Choose two random points defining a segment in the genotype and shuffle the elements in the segment.
- *Displacement*: Choose two random points defining a segment and move the segment to another position in the genotype.
- *Inversion*: Choose two random points defining a segment in the genotype and reverse the order of the segment.
- *Insertion*: Similar to Displacement but only moves one gene.
- *Displacement Inversion*: As the name says choose two random points defining a segment in the genotype, reverse the order and displace it somewhere else.
- *Random*: One of the above is randomly selected.

3.4 Code

The structure of our program is split into four parts. The main.py contains the general outline of the algorithm described at the beginning of Section 3 is used to run the program.

The config.py only contains a set of parameters which set the used methods for selection, crossover etc. and parameters determining a more elitist or exploratory algorithm.

Third we have the organism.py containing the *Organism* class for an object oriented style. It described the inherent structure with genotypes and fitnesses and contains all the used functions for crossover and mutations.

Lastly we have the population.py with the *Population* class which represents a collection of organisms. It also contains methods for sorting the population, selecting parents for mating and more.

We implemented the algorithm in Python 3.6+ and made heavy use of the Numpy package because of its features and ease of use in array manipulation and randomness methods.

4 Results

A typical output of our program for the 8-Queens problem looks like this (if verbose is set to *True*:

```
Fitness: 28.0
Genotype: [3 1 6 2 5 7 0 4]
| | | |Q| | | | |
| |Q| | | | | | |
| | | | | | |Q| |
| | |Q| | | | | |
| | | | | |Q| | |
| | | | | | | |Q|
|Q| | | | | | | |
| | | | |Q| | | |
Number of Iterations:2
Total Time: 0.03384089469909668
Average Fitness of final Population: 24.95

Process finished with exit code 0
```

It shows the fitness, the genotype and a graphical representation of the (or one) optimal solution. Furthermore it shows some data as the running time, the number of iterations and the average fitness of the final population.

For the next sections we wanted to investigate the performance behavior of different selection, crossover and mutation methods. For this we fixed one method (e.g. a fix crossover method) and set the others (e.g. selection and mutation) to random. We have run the benchmark for a 8 and a 15 sized chessboard and computed the average over 500 ($n = 8$) and 100 (n=15) runs for each respectively. If not stated otherwise we set the following parameters:

- Population Size: 100
- Crossover probability: 80 %
- Mutation probability: 30 %
- Copy threshold (to copy a percentage from the old generation to the new one): 10%
- Truncation threshold: 50%
- Tournament competitors: 10

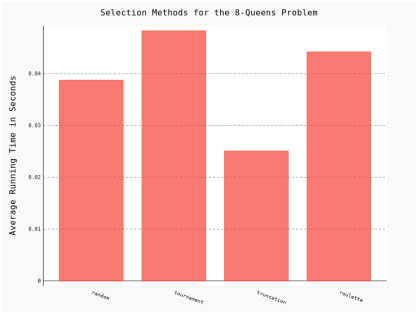
4.1 Selection Methods

In Figure 1 we show the computation time and number of iterations results for $n = 8$ and $n = 15$ for each selection method of *random*, *tournament*, *truncation* and *roulette*.

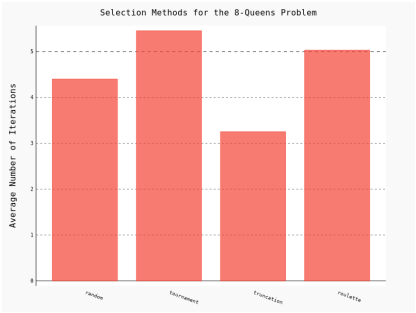
For this benchmark we used only a truncation threshold of 50% and 10 tournament competitors. We will later investigate these values further briefly.

As one can see that for the smaller sized chessboard the truncation method seems to outperform the other ones. But roulette wins for the 15-Queens Problem. In here there is no strong disparity between the computation time and number of iterations which can happen if for example one method produces good results but takes long to compute them. In this case the iterations would decrease but the time would remain high. But neither of them seems the case here.

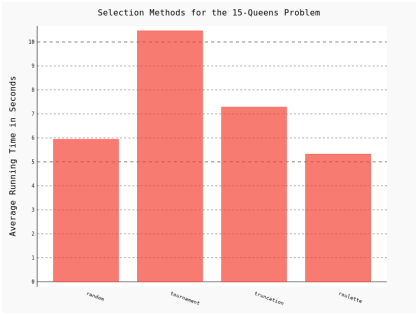
As one would expect *random selection* seems to be a middle ground with taking the third or second place. Maybe it behaves better in some cases because it mixes some things up, as it choses the methods randomly, and there for choses random a more elitist or more exploratory approach.



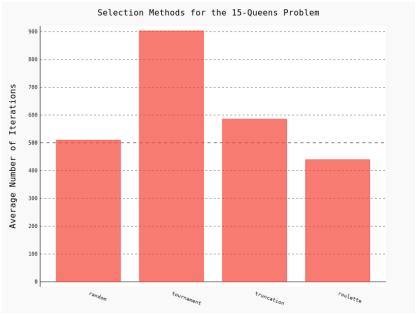
(a) 8-Queens, Time



(b) 8-Queens, Iterations



(c) 15-Queens, Time

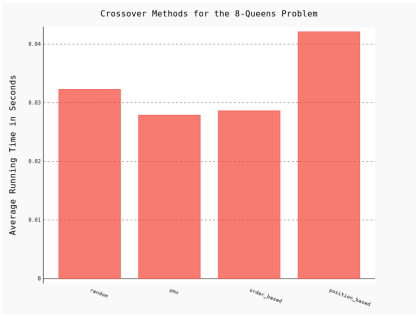


(d) 15-Queens, Iterations

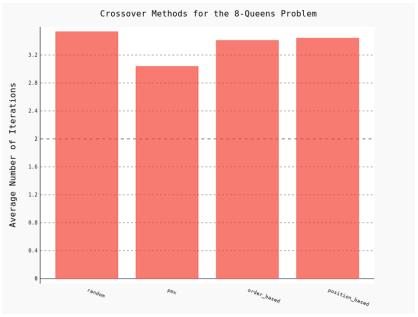
Figure 1: Parent Selection Method Comparison

4.2 Crossover Methods

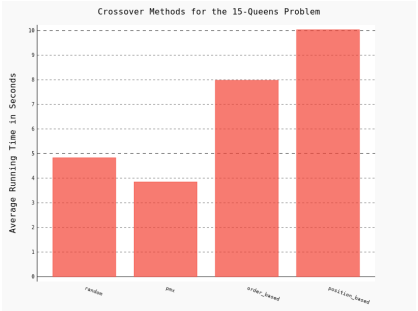
In our benchmarks it is the case that *pmx* is the clear winner in the crossover methods even though the order-based crossover may seem to be competitive for the smaller chessboard. But for $n = 15$ it stands no the chance against the partially mapped crossover.



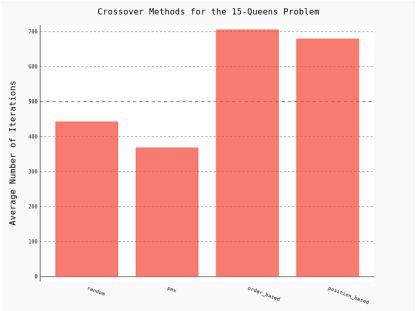
(a) 8-Queens, Time



(b) 8-Queens, Iterations



(c) 15-Queens, Time



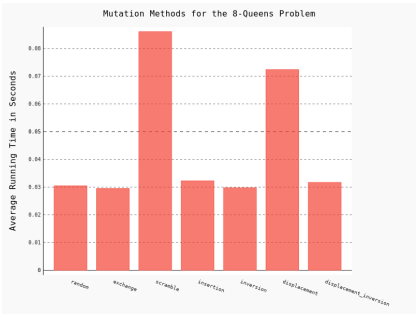
(d) 15-Queens, Iterations

Figure 2: Crossover Method Comparison

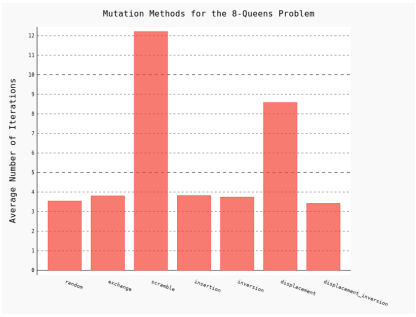
Similar to the selection methods the random crossover is again in the middle and performs a bit better with a higher queen number.

4.3 Mutation Methods

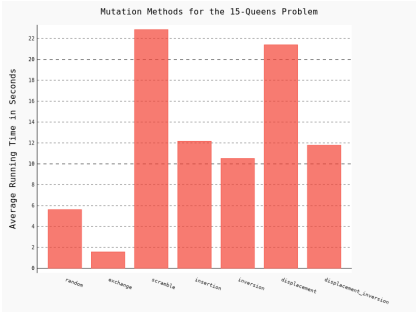
As one can see in Figure 3 there are two clear losers among the mutation methods: *scramble* and *displacement*. The others seem to be equal at first sight but for the bigger chessboard, *exchange* outperforms all the other methods. At least this is true for our case but in [1] it is noted that the insertion should be the best, but that mutations may behave differently for different problems.



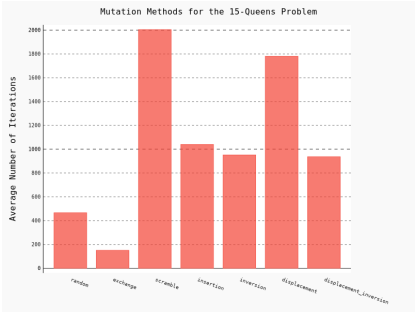
(a) 8-Queens, Time



(b) 8-Queens, Iterations



(c) 15-Queens, Time



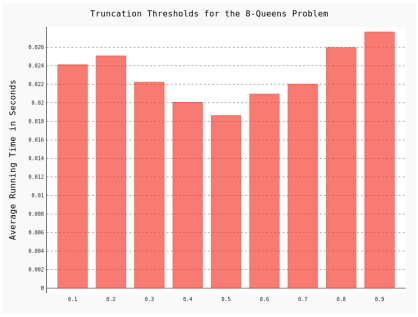
(d) 15-Queens, Iterations

Figure 3: Mutation Method Comparison

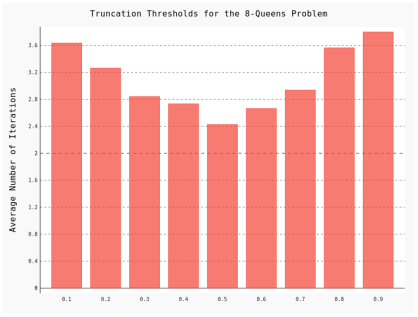
4.4 Truncation Threshold Parameter

For this and the following section we wanted to investigate further the parameters regarding the *truncation* and *tournament* selection. Therefore we decided to take the best mutation and crossover method, which we obtained from the data above. Sol we pick:

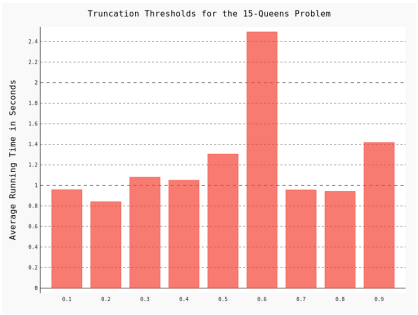
- Crossover Method: Partially matched with crossover probability of 80%
- Mutation Method: Exchange with a mutation probability of 30%



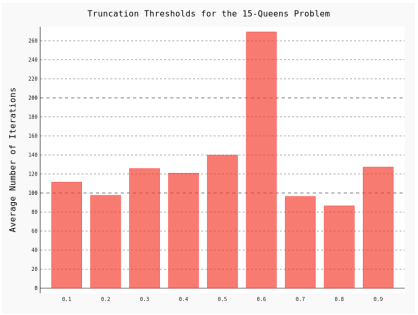
(a) 8-Queens, Time



(b) 8-Queens, Iterations



(c) 15-Queens, Time



(d) 15-Queens, Iterations

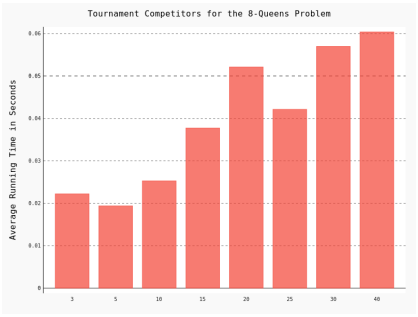
Figure 4: Truncation Threshold Comparison

4.5 Tournament Competitors Parameter

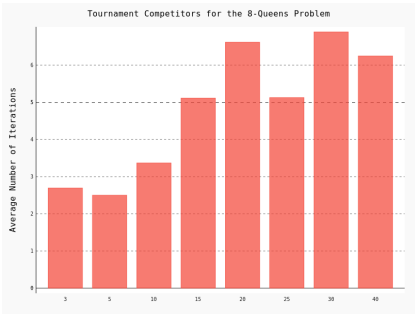
Here we can see (Figure 5) that the best values for the tournament methods are rather small, in particular 5-10 seem to be good values for tournament competitors.

Smaller numbers are more exploratory than elitist because we must remember we pick n competitors randomly and then choose the best of them. The bigger n, the bigger the probability to choose a fit one.

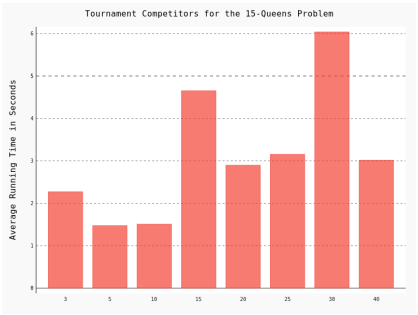
But all in all even with some tuning it does not seem that the tournament method can compete with the truncation method from 4.4



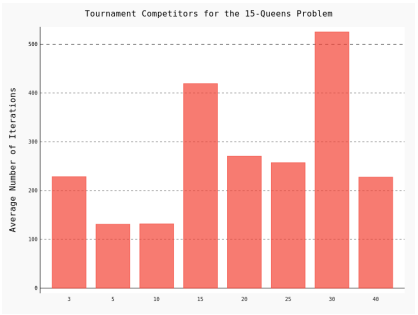
(a) 8-Queens, Time



(b) 8-Queens, Iterations



(c) 15-Queens, Time



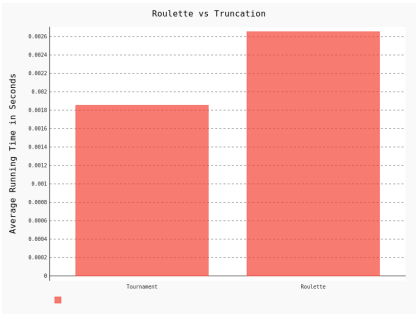
(d) 15-Queens, Iterations

Figure 5: Tournament Competitors Comparison

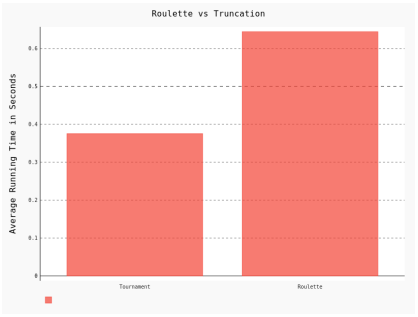
4.6 Solutions for bigger Chessboard Sizes

The best selection method was not really clear from Section 3.1. So we investigated further which one of them performs better for larger n. As a truncation threshold we stick with 50%.

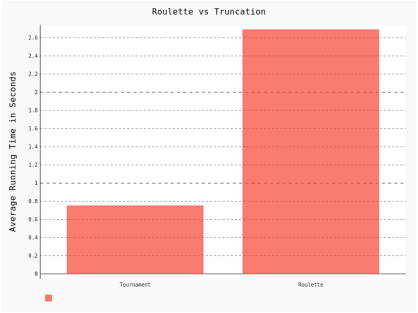
We took again a number of benchmarks and let each selection method run 20 times. Again we only used the partially mapped crossover and exchange mutation.



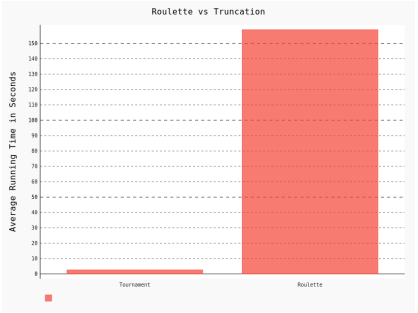
(a) 8-Queens



(b) 15-Queens



(c) 20-Queens



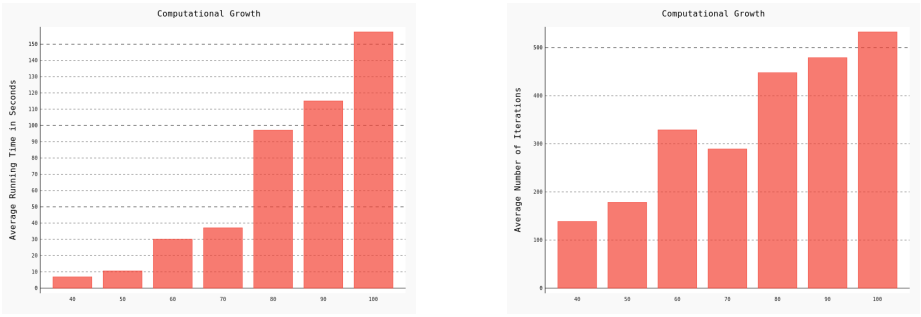
(d) 30-Queens

Figure 6: Comparison for Truncation vs Roulette Selection for bigger Field Sizes.

Now it really seems clear that in particular for bigger field sizes the truncation method is way better than the roulette method. Therefore it would complement our genetic algorithm framework together with the partially mapped crossover and the exchange mutation.

4.7 Computational Growth

The last thing we wanted to show is the exponential complexity of the algorithm/problem and its feasibility even for quite large chessboards. For this test we used our best setting from the benchmarks above: truncation selection, partially mapped crossover and exchange mutation.



(a) Running Time (b) Iterations

Figure 7: Exponential Complexity for increasing Field Sizes

For 100×100 chessboard the algorithm was able to produce reliably optimal solutions with a computation time ranging from 60 to 500 seconds.

5 Conclusion

For the small 8-queens example, we see that there are a few mutation methods that are equally good. Here we choose the exchange method because it is clear winner when the field size gets larger. For the crossover method pmx is clearly the fastest, this is also the case when the field size increases. The best selection method in our case is the truncation method. Here the truncation threshold determines how exploratary (high threshold) or how exploitatory (low threshold) the algorithm will be. Other parameters also have an influence in this, for example the mutation probability.

Besides implementing the algorithm in a faster programming language (like *C*) or interfacing the algorithmic bottlenecks to C through the C Foreign Function Interface (CFFI) there may also be more improvement with different selection, crossover and mutations methods we did not consider so far. Also one can further try to improve and tune the parameters like copy threshold (which we set to 10%) or varying crossover probabilities. Lastly one can to try to improve further the adaptiveness of the algorithm. That is try to detect cases in which the algorithm seems to be stuck in a *local maximum* and be more exploratory.

References

[1] John McCullock, *Genetic Algorithms.*, `http://mnemstudio.org/genetic-algorithms-algorithm.htm`