Universitat Autònoma de Barcelona

Solving 8-Queens Problem with Genetic Algorithms

Research Innovation (Python Report)

Christian Guzman Ruiz *Jeremy J. Williams*

November 2018

Abstract

Genetic Algorithms & their applications have come very useful and especially efficient with optimization problems. Most optimisation problems, theoretical that use **Genetic Algorithms**, follow a path towards a solve search and then optimise criteria. This algorithm replicates the process of '**natural selection**' and '**survival if the fittest**', where the suitable entities are selected for reproduction to produce an offspring for the next generation.

This report focus will be to study a typical **Genetic Algorithm** process that requires a genetic representation and a fitness function to evaluate a solution domain. We want to develop a special instance of the **N-Queens** problem, where the objective is to position **N-Queens** on an **N x N** chessboard such that no two queens can attack each other. The report will be an experimental study with a motivation to successfully assess, evaluate and develop the use of the **selection**, **crossover** and **mutation** methods. And also an "**ad-hoc**" implementation of the special instance of the **N-Queens** problem, where **N = 8**.

The Solution of the **8-Queens** Problem with **Genetic Algorithms** were explained with a focus on **execution times**, **iterations**, **population**, **asymptotic convergence** and **unique solutions**. Results of the **8-Queens** problem discovered were also deliberated.

Contents

1 Introduction

Genetic Algorithms (GA) are adaptive methods which may be used to solve search and optimisation problems. They are based on the genetic processes of biological organisms. Over many generations, natural populations evolve according to the principles of natural selection and "survival of the fittest", first clearly stated by Charles Darwin in The Origin of Species. By mimicking this process, genetic algorithms are able to "evolve" solutions to real world problems, if they have been suitably encoded. For example, GA can be used to design bridge structures, for maximum strength/weight ratio, or to determine the least wasteful layout for cutting shapes from cloth. They can also be used for online process control, such as in a chemical plant, or load balancing on a multiprocessor computer system.[1]

1.1 The problem: 8-queens problem

The 8 queens puzzles is the problem of placing eight chess queens on an **8**×**8** chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general '**N**' queens problem of placing '**N**' non-attacking queens on an **N**×**N** chessboard, for which solutions exist for all natural numbers **N** with the exception of **N** = **2** and **N**=**3**.[2]

The 8 queens puzzle has **92** distinct solutions. For this work, we use GA to obtain one of the possible solutions.

2 Definitions & Concepts

Genetic Algorithms (GA) begins, like any other optimization algorithm, by defining the optimization variables. It ends like other optimization algorithms too, by testing for convergence.

To conduct our study of GA, it is important to declare all essential definitions and concepts.

Define cost function and cost

For each problem there is a cost function. For example, maximum of a 3D surface with peaks and valleys when displayed in variable space. Cost, a value for fitness, is assigned to each solution.

Chromosomes and Genes

A gene is a number between 0 to n-1. A chromosome is an array of these genes. It could be an answer. The Population in each generation determines the number of chromosomes.

Create a random initial population

An initial population is created from a random selection of chromosomes. The number of generations needed for convergence depends on the random initial population.

Decode the chromosome and find the cost

To find the assigned cost for each chromosome a cost function is defined. The result of the cost function called is called cost value. Finally, the average of cost values of each generation converges to the desired answer.

Mating and next generation

Those chromosomes with a higher fitness (lesser cost) value are used to produce the next generation. The off-springs are a product of the father and the mother, whose composition consists of a combination of genes from them (this process is known as "crossing over").

If the new generation contains a chromosome that produces an output that is close enough or equal to the desired answer, then the problem has been solved. If this is not the case, then the new generation will go through the same process as their parents did. This will continue until a solution is reached.

2.1 Art of the Game

In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. However, as the board size increases the game become quite complex. Here illustrate how much the complexity of the game increases with board size.

Looking at a 4 row and 4 columns (**4**x**4**) chess board, a queen (**player 1**) has made it´s intital placement. The gray squares are attacked and the white squares are available to second queen (**player 2**).

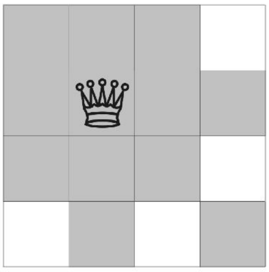


Figure 1: Intital Queen placement on (4x4) chess board from player 1

A (**8**x**8**) chess board has 8 rows and 8 columns. The standard 8 by 8 queen's problem asks how to place 8 queens on an ordinary chess board so that none of them can hit any other in one move. As we can see from understanding the (**4**x**4**) chess board, the black dots (•) in each squares are attacked and there are no available space for another queen. After all queens have a unique safe placement the problem is solved.

Here we shows one of the solutions to the 8-queen problem using GA, which will be explained in the upcoming sections.

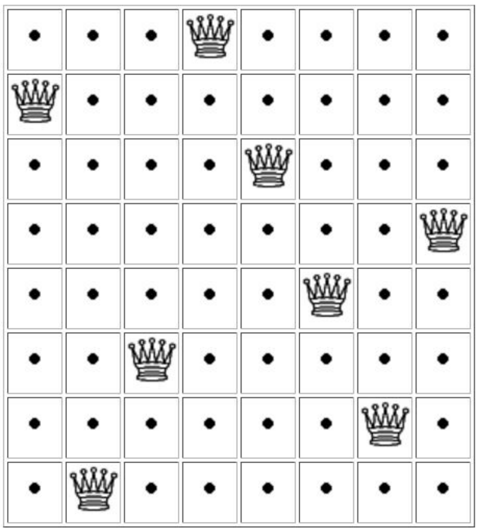


Figure 2: One solution to the 8-queens problem: 6 0 2 7 5 3 1 4

3 Objectives

Our focus will be to study a typical Genetic Algorithm (GA) that requires a genetic representation and a fitness function to evaluate our solution domain. More specifically, we want to develop a special instance of the N-Queens problem, where the objective is to position **N** Queens on an $\mathbf{N} \times \mathbf{N}$ chessboard such that no two queens can attack each other. This special instance in our case is called the '**The Eight (8) Queens Problem**'.

We will develop our GA evolution flow with a given population that provide a possible solution to **The Eight (8) Queens Problem**. Each possibility provides a fitness (cost) value, given by some fitness (cost) function, that represents a best fit value, providing a good configuration solution for the **The Eight (8) Queens Problem**.

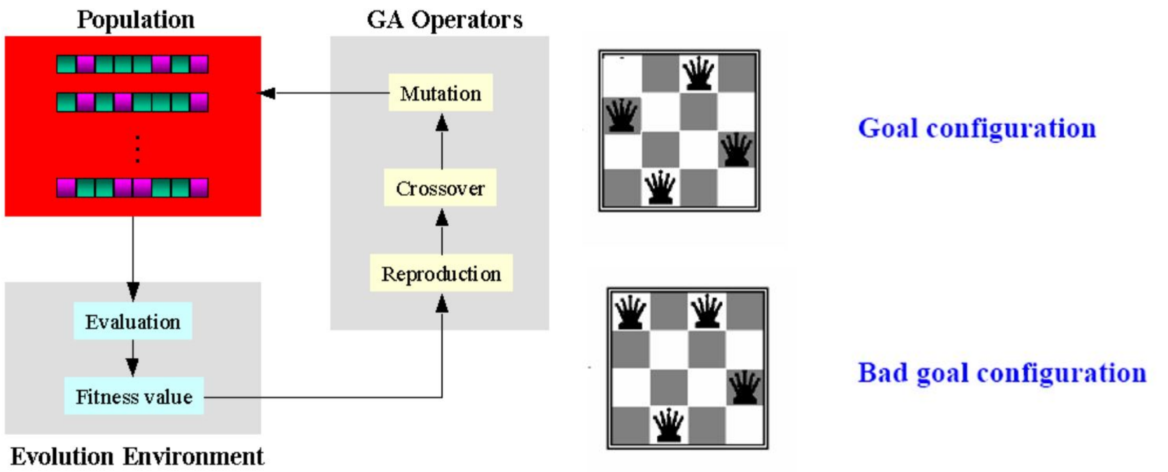


Figure 3: GA Evolution Flow with a "Good" vs "Bad" configuration solution

In this project, we will solve **The Eight (8) Queens Problem** by means of a genetic algorithm developed "**ad-hoc**" and the case of having more queens, i.e **N-Queens**.

We also will pay special attention, in our coding implementation, on three types of basic biological principles:

- **Selection Method:** Selection of a suitable entities to reproduce
- **Crossover Method:** Reproduction between entities
- **Mutation Method:** Random changes in the genetic information carried by an entity

4 The Algorithm

In this section, we present the Genetic Algorithms (GA) used to program the code and some "ad-hoc" implementations for our 8-queens problem are explained. Also, GA methods are defined and their algorithms presented in future subsections.

We based our code in the GA defined in the book "Essentials of Metaheuristics" [3]:

Algorithm 20 *The Genetic Algorithm (GA)*

```
1: popsize ← desired population size          ▷ This is basically λ. Make it even.

2: P ← {}
3: for popsize times do
4:     P ← P ∪ {new random individual}
5: Best ← □
6: repeat
7:     for each individual P_i ∈ P do
8:         AssessFitness(P_i)
9:         if Best = □ or Fitness(P_i) > Fitness(Best) then
10:            Best ← P_i
11:    Q ← {}                                   ▷ Here's where we begin to deviate from (μ, λ)
12:    for popsize/2 times do
13:        Parent P_a ← SelectWithReplacement(P)
14:        Parent P_b ← SelectWithReplacement(P)
15:        Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
16:        Q ← P ∪ {Mutate(C_a), Mutate(C_b)}
17:    P ← Q                                    ▷ End of deviation
18: until Best is the ideal solution or we have run out of time
19: return Best
```

Figure 4: The GA algorithm in pseudocode

The Algorithm starts initializing parents and end condition **(lines 2-5)**. Then main loop starts, fitness for every parent are calculated and end condition (best) is updated with best parent fitness.

To breed, we begin with and empty population of children. We then select two parents from the original population **(lines 13,14)**, copy them, cross them over with one another **(line 15)** and mutate the results **(line 16)**. This forms two children, which we then add to the child population. We repeat this process until the child population is entirely filled **(line 12)**. We then update parents with children values **(line 17)** and repeat the algorithm till we reach best fitness **(line 18)**.

4.1 Ad-hoc Implementations

A possible representation for the board can be a **N**x**N** matrix, where **N** is the number of queens (eight for our main case). Then, board position with a queen can be represented as value "1" and either (no queen on the position) as value "0". This representation is equivalent to boolean vector, allowing the application of corresponding methods like one-point crossover or bit-flip mutation.

But we prefer to use another representation, more optimum for data memory and iterations. Since queens shouldn't collide between rows and columns, we can represent the queen positions on chess board as an array, where index of the array is the board row and the values are the column positions. This implementation secure queens only that can collide between diagonals. We also notice that the GA methods that are available changes, allowing now only methods for array integers like order crossover (OX) or inversion mutation.

In fact, some of the methods available for our array structure are explained in subject notes. We tested some of them in order to analyze a bit the impact of different methods.

We have to mention some nomenclature. For our problem, parent/individual is board array representation, population refers to all the parents, GA generations are named as iterations and fitness is the negative number of queen collisions on board. For example, if one individual has only one queen in the same diagonal as another, it means two collisions (one for each queen in same diagonal), and fitness will be "-2". GA will end when any individual reach 0 fitness.

4.2 Selection Methods

This operator selects chromosomes in the population for reproduction. The fitter the chromosome the more times it is likely to be selected to reproduce.

There are many types of selection. We tested the following methods: **Fitness-Proportionate selection** and **Tournament**.

Algorithm 30 *Fitness-Proportionate Selection*

perform once per generation
    global $\vec{p} \leftarrow$ population copied into a vector of individuals $\langle p_1, p_2, ..., p_l \rangle$
    global $\vec{f} \leftarrow \langle f_1, f_2, ..., f_l \rangle$ fitnesses of individuals in $\vec{p}$ in the same order as $\vec{p}$
    if $\vec{f}$ is all 0.0s **then** ▷ Deal with all 0 fitnesses gracefully
        Convert $\vec{f}$ to all 1.0s
    for i from 2 to l **do** ▷ Convert $\vec{f}$ to a CDF. This will also cause $f_l = s$, the sum of fitnesses.
        $f_i \leftarrow f_i + f_{i-1}$
perform each time
    $n \leftarrow$ random number from 0 to f_l inclusive
    for i from 2 to l **do** ▷ This could be done more efficiently with binary search
        if $f_{i-1} < n \leq f_i$ **then**
            return p_i
    return p_1

Figure 5: Fitness-Proportionate selection algorithm

4.3 Crossover Methods

This operator randomly chooses a part from individual and exchanges the subsequences before and after the locus between two chromosomes to create two offspring. The cros-

Algorithm 32 *Tournament Selection*

```
P ← population
t ← tournament size, t ≥ 1

Best ← individual picked at random from P with replacement
for i from 2 to t do
    Next ← individual picked at random from P with replacement
    if Fitness(Next) > Fitness(Best) then
        Best ← Next
return Best
```

Figure 6: Tournament Algorithm

sover operator roughly mimics biological recombination between two single chromosome (haploid) organisms.

There are many types of crossover. We tested the following methods: **Order (OX) crossover** and **Position-Based**.

Procedure: OX

1. Select a substring from a parent at random.
2. Produce a proto-child by copying the substring into the corresponding position of it.
3. Delete the cities which are already in the substring from the 2nd parent. The resulted sequence of citires contains the cities that the proto-child needs.
4. Place the cities into the unfixed positions of the proto-child from left to right according to the order of the sequence to produce an offspring.

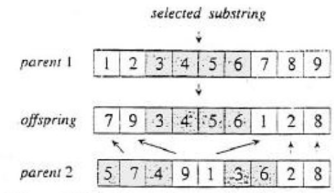


Figure 7: Illustration of OX Operator

Procedure: Position-Based Crossover
1. Select a set of position from one parent at random.
2. Produce a proto-child by copying the cities on these positions into the corresponding position of the proto-child.
3. Delete the cities which are already selected from the second parent. The resulting sequence of cities contains the cities the proto-child needs.
4. Place the cities into the unfixed position of the proto-child from left to right according to the order of the sequence to produce one offspring.

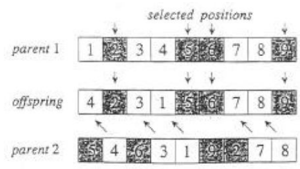


Figure 8: Illustration of Position-Based Operator

4.4 Mutation Methods

This operator randomly flips some of the bits in a chromosome. Mutation can occur at each bit position in a string with some probability, usually very small.

There are many types of mutation. We tested the following methods: **inversion**, **insertion** and **exchange** mutation.

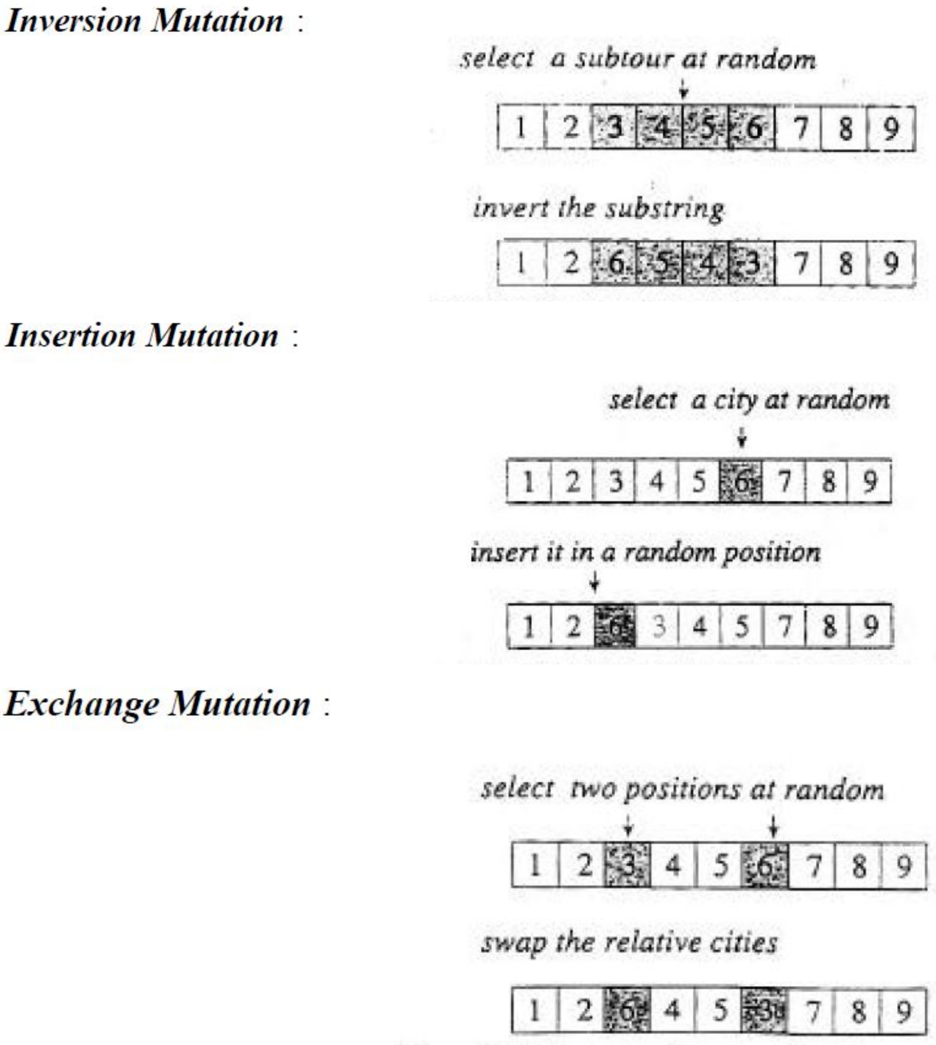


Figure 9: Illustration of inversion, insertion and exchange mutation respectively

5 Code Implementation

In this section, a presentation of an explanation of the Genetic Algorithm in **Python** code used to solve the 8 queens problem. We present all important variables and functions. The complete code can be found at appendix.

5.1 Class: GAQueen

Main class of the project. Contains GA functions and variables like population size, parents, children and fitness.

5.2 Variables: Structure of data

We explain only most important variables, which corresponds to structure of main data:

boards[population][N]:

Population size per queens number matrix. Correspond to GA parents.

childboards[population][N]:

Population size per queens number matrix. Correspond to GA children.
Children are only different from parents during methods loop. After methods loop end, childboards content will be copy into parents.

fitness[population]:

Population size array. Fitness index are related to parents index. This means, for example, first fitness element return first parent (board[0]) fitness.

5.3 Functions

initParent(self, a):

Assign range(N) elements randomly ordered to parent and child a.
param a: Index of parent

assessFitness(self, a):

Check the number of collisions between queens. We only have to check diagonal collisions since queens are in different rows and columns every time. Less collisions means best fitness, with maximum fitness of 0.
param a: Index of parent

selectWithReplacement(self,method):

Returns Parent in population according to the chosen method.

param method: Method selected

return: Index of parent selected

crossover(self,method,a,b):

Crossover two parents of boards population and update childBoards according to the chosen method. Position-based and order

param method: Method selected param a: Index of first parent param b: Index of second parent param c: Index of first child

mutate(self,method,a):

Mutate a childBoards according to the chosen method

param method: Method selected param c: Index of a childBoards

main():

This function input the arguments of population, iteration and execution, either use default values. Then, create a GAQueen object and initialise the following member variables: population size, maximum iterations and number of executions.

After, initialise local variables. The most important are corresponding to selection, crossover and mutation methods. Then, GA starts initialising population and resetting best fitness variable. Follow entering in main loop, where methods are applied over population till some individual reach best fitness (0) or maximum iterations. After this, GA repeats for execution number.

Finally, print statistics (explained at results section) and parent with best fitness results (like figure 8) to verify problem is resolved correctly.

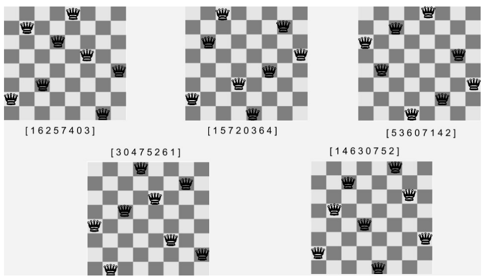


Figure 10: Genetic Algorithms Solutions and Chessboard Locations

6 Results

We will check what method is more efficient in terms of time and generations (we will name "**generations**" as "**iterations**" for now) for our problem. For 8 queens problem the execution time is very low, and can be affected notably by noise distortions. To fix we also check the number of iterations. Notice that iterations and time execution should be **correlated**.

Also notice high population will result in low iterations, maybe not enough to test correctly the methods and low population can also be insufficient to analyze the effect of methods complexity on time execution. We interesting are having an intermediate number of iterations to study the effect of the methods. We also will check the effect of population variation for our problem and define the number of population used to test the methods.

6.1 Population

We used the following initial configuration for the population results. Also, all the results are averaged over **500** executions. Time units are seconds.

selectionMethod = **tournament**

crossoverMethod = **order**

mutationMethod = **inversion**

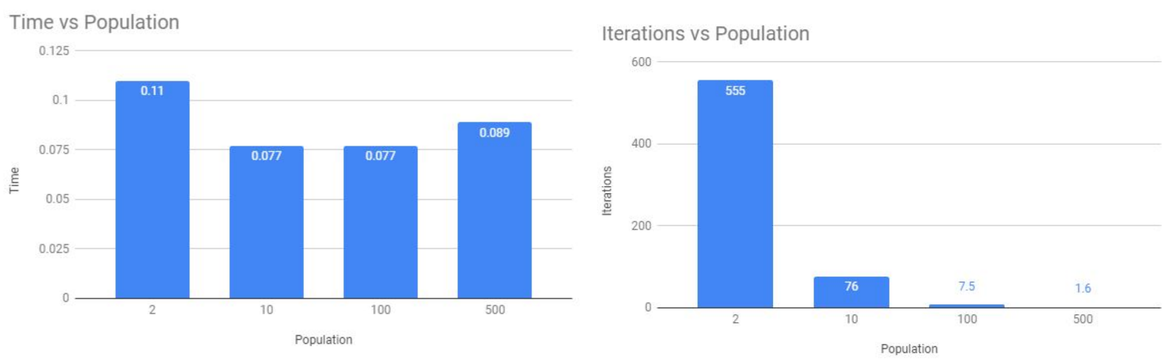


Figure 11: Graphics of execution time, iterations and population

We can see how the time is very similar for 10 and 100 population cases, and notably more for 2 and 500 population cases. This means GA works well with balanced population/iteration rate, which is logical, since GA do not take full advantage of the methods for low iterations or mutations.

We can also see the iterations decreases proportionally to population, for example, for 10 population to 100 population (x10 scale) iteration decreases at the same proportion (10x scale).

Moreover, we can select the population to test the methods by checking the number of iterations. We consider that we need more than 10 iterations to test correctly the different methods and also a population more than 2 to test correctly the selection methods. Then we decide to choose 10 population size for the methods test explained in next sections.

We tested also the problem using 10 queens and 10x10 board size. We have to mention that we only average over 10 executions to avoid high time executions. The noise over the same configuration is high, but checking at the results we can get some conclusions:

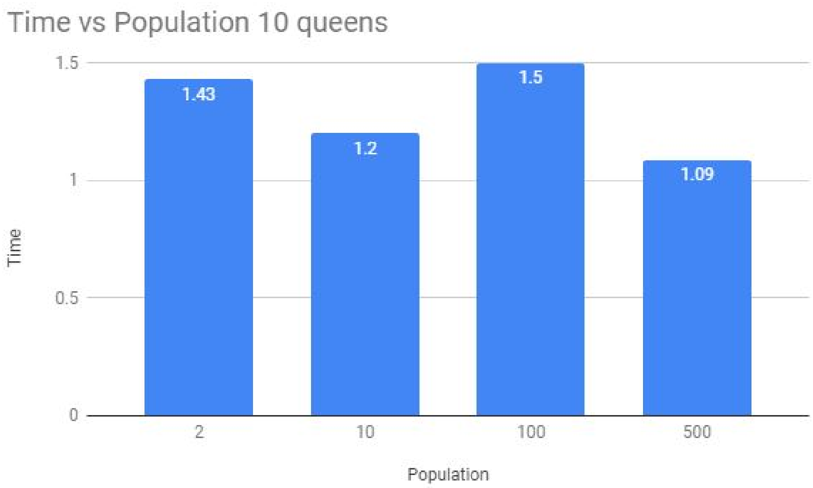


Figure 12: Graphics of execution time and population for 10 queens

However, we can conclude that the proportion are very similar (like 8 queens) and the time difference between population are invariant of the number of queens.

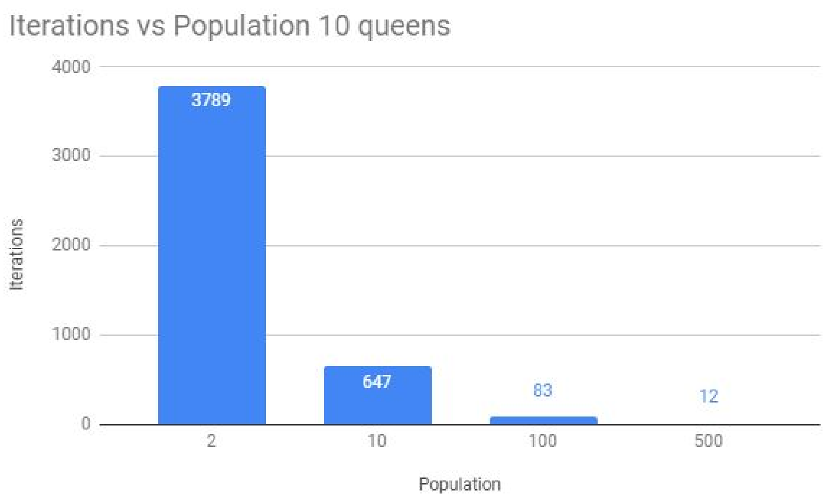


Figure 13: Graphics of iterations and population for 10 queens

For iterations, population and iteration are proportional at similar rate as 8 queen problem (10 to 100 population decreases near x10 iterations). Also, we can observe that number of iterations rises with the augment of queens, making high population (100 or 500) viable to study (they have more than 10 iterations to notice the methods influence).

We can conclude low population is bad, and get worse for large problems. Population and iterations should be on a balanced, without having too few population or iterations.

6.2 Selection Methods

Testing the selection methods we obtained the following results:

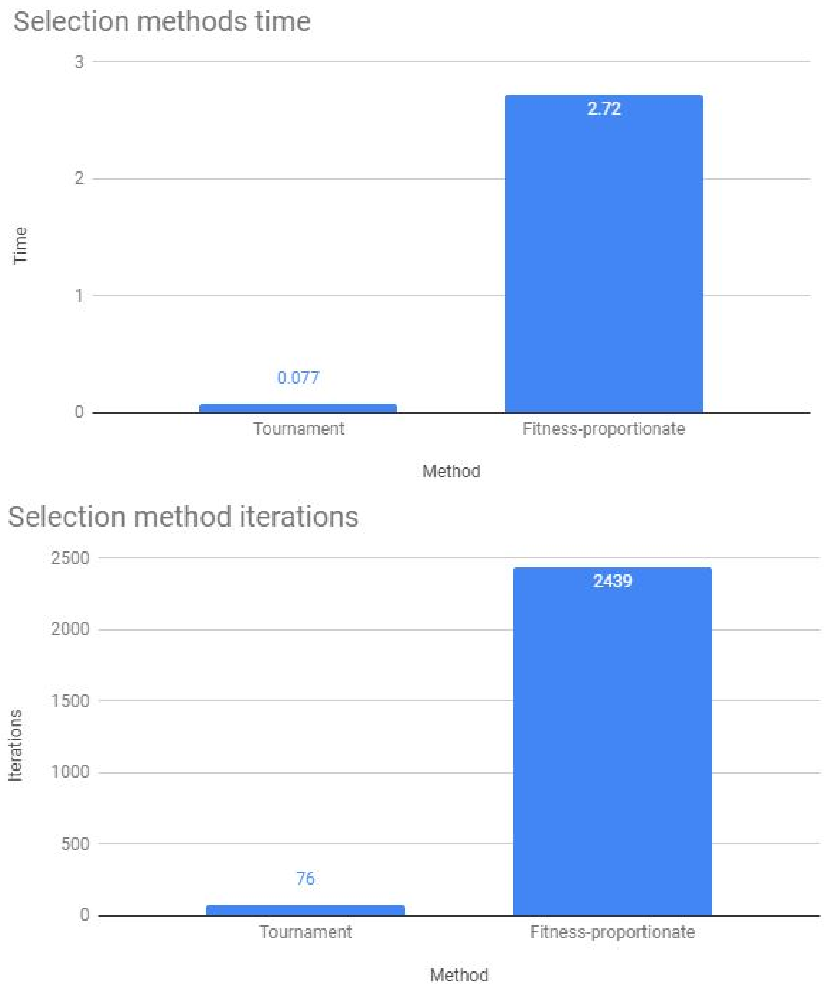


Figure 14: Graphics of time and iterations for selection methods

We can see how the tournament selection are providing better results than fitness-proportionate selection. This means the **Cumulative Distribution Function** (CDF) calculation of fitness-proportionate is very expensive in time execution.

6.3 Crossover Methods

Testing the crossover methods we obtained the following results:

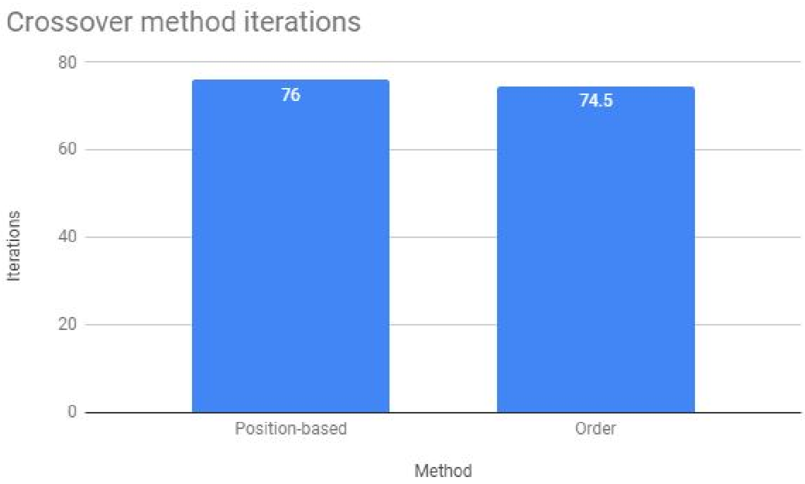


Figure 15: Graphics of time and iterations for crossover methods

Both methods has the same number of iterations, but different times, specifically, position-based looks a bit better than order.

6.4 Mutation Methods

Testing the mutation methods we obtained the following results:

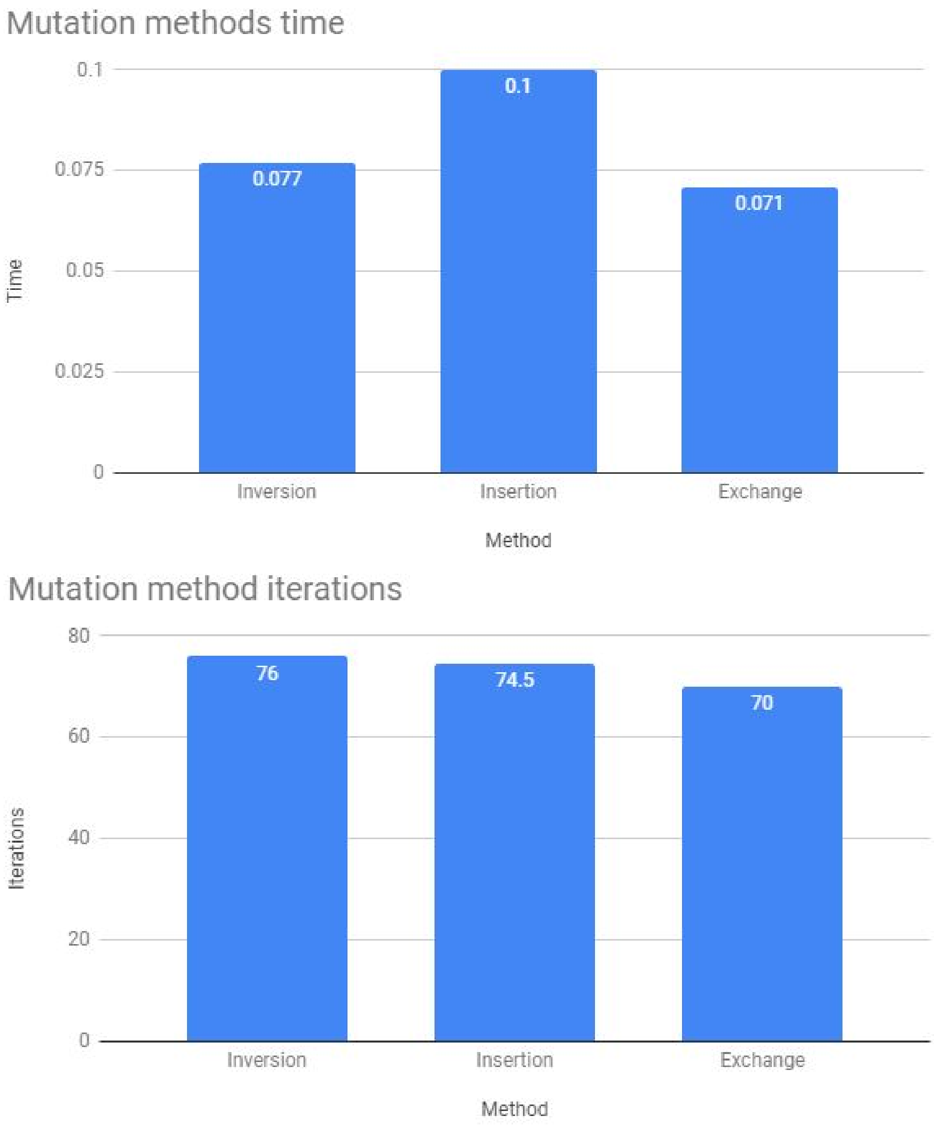


Figure 16: Graphics of time and iterations for mutation methods

All methods has nearly the same iterations, and insertion is a bit slower than other methods. Inversion and exchange have the same time.

6.5 Asymptotic convergence

We also check the asymptotic convergence of the best methods combination (tournament, position-based and exchange) and 10 population size.

We tested the problem for 8 and 10 queens obtaining the following results:

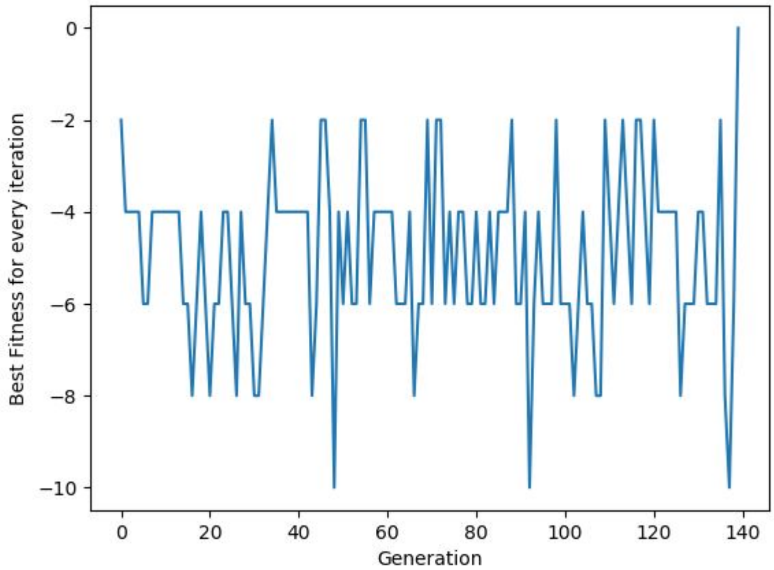


Figure 17: Graphic of asymptotic convergence for 8 queens, 10 population size and best methods combination

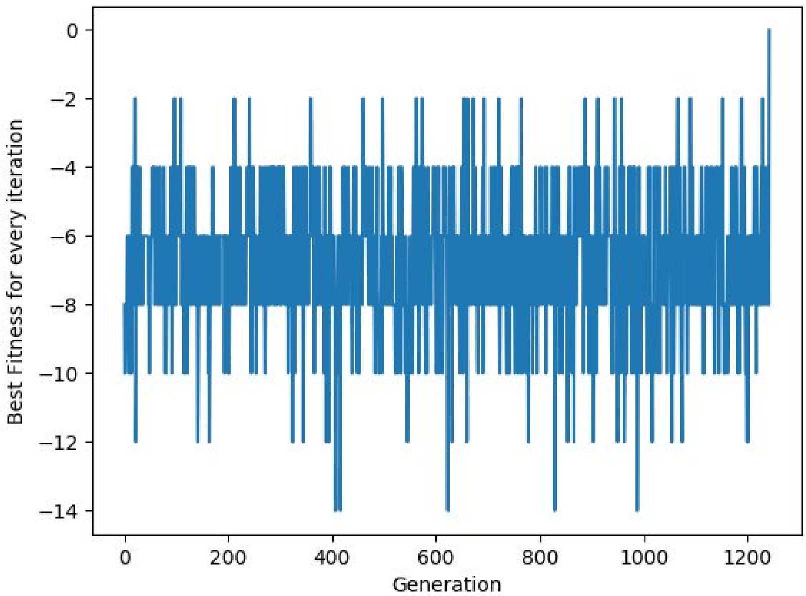


Figure 18: Graphic of asymptotic convergence for 10 queens, 10 population size and best methods combination

Both graphics have a similar graph response. Notice that they reach the second best fitness (-2) more times than worst fitness for each graphic (-10, -14), which is a good symptom.

7 Solutions

In this section, we present an additional modification to our Genetic Algorithms (GA) implementation that was designed to get all solutions to the special instance N-Queens Problem. We will also provide additional cases for more number of Queens.

7.1 Queens Solution Results

This is usual output of our solution generating program for the N-Queens problem to a .CSV file (when N = 8):

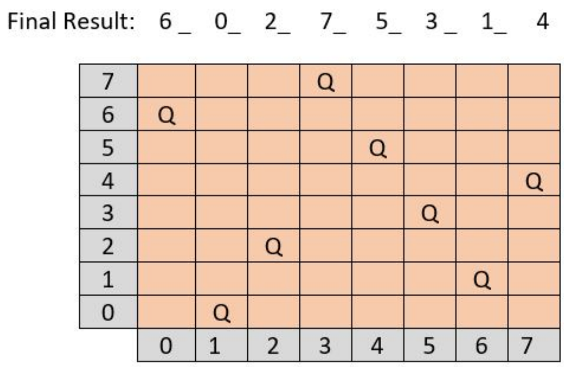
```

Final result:
6_0_2_7_5_3_1_4

All solutions:
0_4_7_5_2_6_1_3
0_5_7_2_6_3_1_4
0_6_3_5_7_1_4_2
0_6_4_7_1_3_5_2
1_3_5_7_2_0_6_4
1_4_6_0_2_7_5_3
1_4_6_3_0_7_5_2
1_5_0_6_3_7_2_4
1_5_7_2_0_3_6_4
1_6_2_5_7_4_0_3
1_6_4_7_0_3_5_2
1_7_5_0_2_4_6_3
2_0_6_4_7_1_3_5
2_4_1_7_0_6_3_5
2_4_1_7_5_3_6_0
2_4_6_0_3_1_7_5
2_4_7_3_0_6_1_5
2_5_1_4_7_0_6_3
2_5_1_6_0_3_7_4
2_5_1_6_4_0_7_3
2_5_3_0_7_4_6_1
2_5_3_1_7_4_6_0
2_5_7_0_3_6_4_1
2_5_7_0_4_6_1_3
2_5_7_1_3_0_6_4
2_6_1_7_4_0_3_5
2_6_1_7_5_3_0_4
2_7_3_6_0_5_1_4
3_0_4_7_1_6_2_5
3_0_4_7_5_2_6_1
3_1_4_7_5_0_2_6
3_1_6_2_5_7_0_4
3_1_6_2_5_7_4_0
3_1_6_4_0_7_5_2
3_1_7_4_6_0_2_5
```

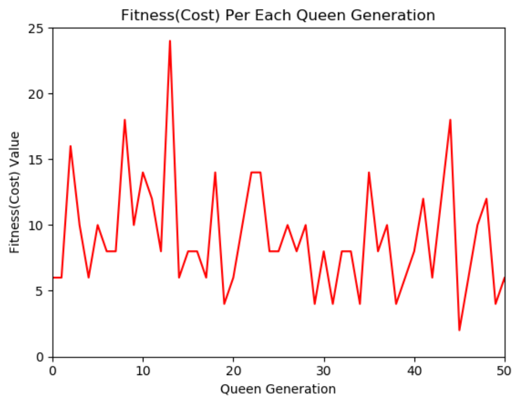
```
3_1_7_5_0_2_4_6
3_5_0_4_1_7_2_6
3_5_7_1_6_0_2_4
3_5_7_2_0_6_4_1
3_6_0_7_4_1_5_2
3_6_2_7_1_4_0_5
3_6_4_1_5_0_2_7
3_6_4_2_0_5_7_1
3_7_0_2_5_1_6_4
3_7_0_4_6_1_5_2
3_7_4_2_0_6_1_5
4_0_3_5_7_1_6_2
4_0_7_3_1_6_2_5
4_0_7_5_2_6_1_3
4_1_3_5_7_2_0_6
4_1_3_6_2_7_5_0
4_1_5_0_6_3_7_2
4_1_7_0_3_6_2_5
4_2_0_5_7_1_3_6
4_2_0_6_1_7_5_3
4_2_7_3_6_0_5_1
4_6_0_2_7_5_3_1
4_6_0_3_1_7_5_2
4_6_1_3_7_0_2_5
4_6_1_5_2_0_3_7
4_6_1_5_2_0_7_3
4_6_3_0_2_7_5_1
4_7_3_0_2_5_1_6
4_7_3_0_6_1_5_2
5_0_4_1_7_2_6_3
5_1_6_0_2_4_7_3
5_1_6_0_3_7_4_2
5_2_0_6_4_7_1_3
5_2_0_7_3_1_6_4
5_2_0_7_4_1_3_6
5_2_4_6_0_3_1_7
5_2_4_7_0_3_1_6
5_2_6_1_3_7_0_4
5_2_6_1_7_4_0_3
5_2_6_3_0_7_1_4
5_3_0_4_7_1_6_2
5_3_1_7_4_6_0_2
5_3_6_0_2_4_1_7
5_3_6_0_7_1_4_2
5_7_1_3_0_6_4_2
6_0_2_7_5_3_1_4
6_1_3_0_7_4_2_5
6_1_5_2_0_3_7_4
6_2_0_5_7_4_1_3
6_2_7_1_4_0_5_3
6_3_1_4_7_0_2_5
6_3_1_7_5_0_2_4
```

```
6_4_2_0_5_7_1_3
7_1_3_0_6_4_2_5
7_1_4_2_0_6_3_5
7_2_0_5_1_4_6_3
7_3_0_2_5_1_6_4

Total: 92 solutions

Total number of queens: 8

DONE!!!

Execution time: 6.477712631225586 seconds

Date: Sat Nov 24 20:55:39 2018
```

7.2 Queens Solution Visualisation

Here we provide a graphical representation of our solution generating program and program plot from our the special instance N-Queens Problem (when N = 8). Also we provide a solutions table results of up to 14 Queens.





No. of Queens	One Solution (with Python Code)	Execution Time (seconds)	Solution Set Size (n!)	No. of Solutions
1	0	0.000000000000000	1	1
2	1_0	17.507363557815500	2	0
3	2_0_1	18.091936826705900	6	0
4	2_0_3_1	1.873886823654170	24	2
5	1_4_2_0_3	1.842944145202630	120	10
6	2_5_1_4_0_3	7.508281707763670	720	4
7	4_2_0_6_3_5_1	16.438174962997400	5040	40
8	**6_0_2_7_5_3_1_4**	**6.477712631225580**	**40320**	**92**
9	3_6_4_1_8_0_5_7_2	9.451443672180170	362880	352
10	8_3_1_4_7_5_0_2_9_6	19.114209413528400	3628800	724
11	1_9_3_8_10_2_0_5_7_4_6	21.607742309570300	39916800	2680
12	8_6_1_7_11_0_3_10_5_2_4_9	35.595991849899200	479001600	14200
13	9_7_5_0_1_10_4_6_8_11_2_12_3	136.705286026000000	6227020800	73712
14	8_4_13_3_10_12_1_6_2_7_11_0_9_5	1366.790939331050000	87178291200	365596

Figure 19: Queens location on a NxN board when N = 8, 50 Generations when N = 8 and Solution Results Table of up to 14 Queens.

8 Discussion and Conclusions

8.1 Discussion

In this report, we have developed a genetic algorithm to solve a special instance of the **N-Queens** problem, where **N = 8**. It has been assumed that **selection**, **crossover**, and **mutation** operators of the genetic algorithm will converge over sequential generations towards achieving best individuals from population.

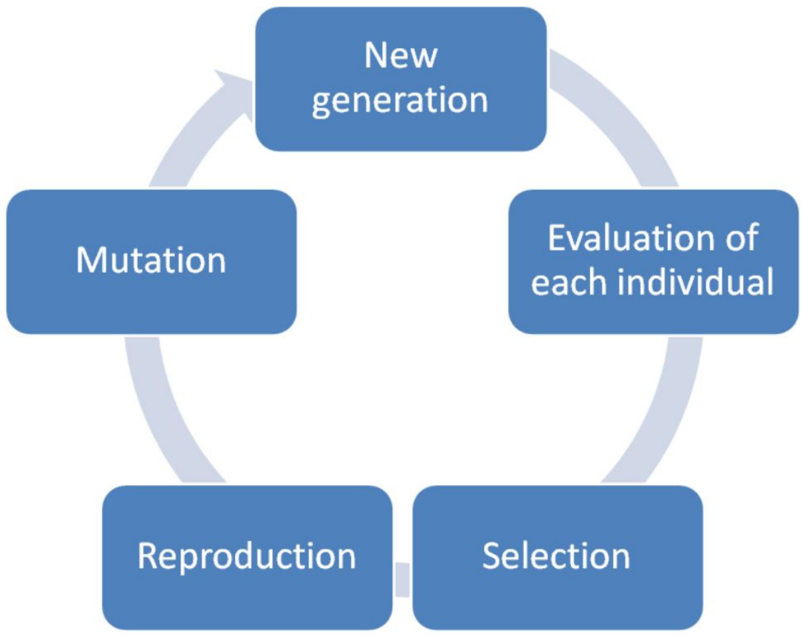


Figure 20: A Genetic Algorithm process over continuous sequential generations

We were able to solve the **8-Queen** problem using two approaches, **1)** a python implementation that provide result based on our domain criteria with a focus on examining the different methods (i.e. selection, crossover and mutation etc.) and **2)** a python modification of our first implementation that provide results based on optioning all unique solutions from a predetermined N value (i.e. **N = 8** showing **92** solutions).

Population

Looking at both methods, low population values are not very useful and tends to increase negatively for large value problems. To ensure good long term performance, it is recommended to have a good balance between the population and iteration values.

Selection

When developing and studying the selection methods process, it can be seen from the original tournament algorithm structure that it is more simpler method. This theory has been proven within our results and execution time showing that the fitness-proportionate selection method is computationally very expensive.

Crossover

As we look that both our chosen methods, there is not a big difference in the result however, what is expected are time spans are different and the position-based methods seems to be preforming better. This can maybe be happening in the deleting part of the procedure.

Mutation

This method have a lower insertion than other methods. We can also see similar iterations exist with similar execution times.

Asymptotic convergence

The asymptotic convergence combination (tournament, position-based and exchange) of the best methods presents the best way forward towards future improvements. The asymptotic convergence graphical representation of problem for 8 and 10 queens problem, keeping population size constant at 10, provide good indicating results of the second best fitness being (-2) more times than worst fitness for each graphic (-10, -14). This shows us that our methods implementation provides normalised and stable results.

Solutions

Within the process after results, we provide a graphical representation of our solution generating program and program plot from our the special instance N-Queens Problem (when N = 8). Also we provide a solutions table results of up to 14 Queens.

8.2 Conclusion

It has been seen that, we have noticed the associations of each process made when designing a Genetic Algorithm implementation. Overall, the performance can change drastically with differently level of populations and iterations; which can become very computationally expensive when an implementation interacts with a mutation process.

Notwithstanding this hindrance, prospective exploration should be concentrated on the level of parameters need to execute any Genetic Algorithm implementation, such as the population size, number of iterations and mutation level that can make a tremendous difference achieving positive performance analysis.

References

[1] D. Beasley, D. R. Bull and R. R. Martin. An Overview of Genetic Algorithms: Part 1, Fundamentals. *University Computing*, 15(2):56–69, 1993.

[2] E. J. Hoffman. *Construction for the Solutions of the m Queens Problem.* Mathematics Magazine, xx edition, 1969.

[3] S. Luke. *Essentials of Metaheuristics.* Lulu, 2nd edition, 2013.

[4] M. Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, 1996.

[5] J. McCullock. Genetic Algorithms. URL: `http://mnemstudio.org/genetic-algorithms-algorithm.htm`.

[6] T. P.I. P. Ltd. Genetic Algorithms Tutorial. URL: `https://www.tutorialspoint.com/genetic_algorithms`.

[7] L. Alsedà. Optimisation: Genetic Algorithms. URL: `http://mat.uab.cat/~alseda/MasterOpt`.

[8] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., 1st edition, 1989.

[9] O. Kramer. *Genetic Algorithm Essentials.* Springer Publishing Company, 1st edition, 2017.

[10] D. Bill. The N-Queens Problem. URL: `http://www.durangobill.com/N_Queens.html`.

Appendices

A1: Python Code for 'N' Queens Methods

```python

import numpy as np
import sys
import random
import matplotlib.pyplot as plt
import time
import math

"""
Ad-hoc suposition: Since queens can't collide between rows and columns, we can reinterpret the queen positions on board
as an array, where index of the array is the board row and the values are the column positions.
Then, we only need to obtain the correct arrangements (solutions of N queens)
"""

start_time0 = time.time() # Init global time counter
N = 8 # Number of queens

class GAQueen:
    def __init__(self, population):
        self.population = population
        self.boards = np.zeros((population, N))
        self.childBoards = np.zeros((population, N))
        self.fitness = np.empty(population)
        self.fitness.fill(-N * (N - 1) / 2)# Maximum number of collisions

    def initParent(self, a):
        """
            Assign range(N) elements randomly ordered to parent and child a
            :param a: Index of parent
        """

        numbers=list(range(N)) #Generate N numbers from 0 to N-1 (the N values for N positions)
        random.shuffle(numbers) #Reorder numbers list randomly
        self.boards[a,:]=numbers #Assign to board
        self.childBoards[a, :] = numbers #Assign to children

    def assessFitness(self, a):
        """
            Check the number of collisions between queens. We only have to check diagonal collisions
            since queens are in different rows and columns every time. Less collisions means best
            fitness, with maximum fitness of 0
```

```
            :param a: Index of parent
        """

        collisions=0 #Reset collisions
        board= self.boards[a,:] # Get board parent

        # check diagonal collisions
        for i in range(N):
            for j in range(N):
                if(i!=j):
                    deltaRow= abs(i - j)
                    deltaCol= abs(board[i]-board[j])
                    if deltaRow == deltaCol: collisions+=1

        self.fitness[a]=collisions*(-1)

    def selectWithReplacement(self,method):
        """
            Returns Parent in population according to the chosen method
            :param method: Method selected
            :return: Index of parent selected
        """

        if method == 'fitness-proportionate':
            fitness = self.fitness.copy()
            n = random.randrange(0, self.population)
            best=-1
            for i in range (1,self.population):
                fitness[i]=self.fitness[i]+self.fitness[i-1] #CDF of fitnesses. Fitness[population] = the sum of fitnesses
                if fitness[i-1] < n <= fitness[i]:
                    best=i
            if best==-1 : best=self.population-1 #If best was not modified, it will be last index

        elif method == 'tournament':
            t=2 #Tournament size, most popular size is 2
            best = random.randrange(0, self.population)
            for j in range (1,t):
                next = random.randrange(0, self.population)
                if self.fitness[next] > self.fitness[best]:
                    best = next

        else: # default: Complete random selection, no selection by fitness
            best = random.randrange(0, self.population)

        return best

    def crossover(self,method,a,b,c):
        """
            Crossover two parents of boards population and update childBoards according to the chosen method
```

```python
            :param method: Method selected
            :param a: Index of first parent
            :param b: Index of second parent
            :param c: Index of first child
        """

        d= c+int(self.population/2) #Index of second child
        self.childBoards[c] = self.boards[a] # Copy parent to child
        self.childBoards[d] = self.boards[b] # Copy parent to child

        if method == 'position-based':
            #Note order-based is the inverse of position-based, and it's done also in this method to get the other child

            for z in range(0, 2): # For the two children
                fixedValues = random.sample(list(range(N)), int(N / 2)) # Fixed values
                nonFixedList = list() # Init "long" substring (in case N uneven) from parent2
                for j in range(0, N):
                    if self.boards[b][j] not in fixedValues:
                        nonFixedList.append(self.boards[b][j]) # Get non fixed elements from parent2
                for j in range(0, N):
                    if self.childBoards[c][j] not in fixedValues:
                        self.childBoards[c][j] = nonFixedList.pop(0) # Get non fixed elements from parent2
                b, a, c = a, b, d # Swap index to generate the other child

        else: # default: order
            for z in range (0, 2):

                start = random.randrange(0, math.ceil(N / 2)) # start of fixed substring (index included)
                end = start + int(N / 2) # End of fixed substring (index not included)

                nonFixedList = list() # Init "long" substring (in case N uneven) from parent2
                for j in range(0, N):
                    # If element in b is not in fixed substring [from start through end-1, where maximun end is equal to N]
                    if self.boards[b][j] not in self.boards[a][start:end]:
                        nonFixedList.append(self.boards[b][j]) # Get non fixed elements from parent2

                # Assign non fixed elements to childBoards "holes"
                for i in range(0, start): self.childBoards[c][i] = nonFixedList.pop(0)
                for i in range(end, N): self.childBoards[c][i] = nonFixedList.pop(0)
                b, a, c = a, b, d # Swap index to generate the other child

```

```
    def mutate(self,method,c):
        """
            Mutate a childBoards according to the chosen method
            :param method: Method selected
            :param c: Index of one childBoards
        """
        if method == 'inversion':
            start = random.randrange(0, math.ceil(N / 2)) # start of fixed substring (index included)
            end = start + int(N / 2) # End of fixed substring (index not included)
            substringReversed = self.childBoards[c][start:end][::-1].copy()
            j = 0
            for i in range(start, end):
                self.childBoards[c][i] = substringReversed[j]
                j += 1

        elif method == 'insertion':
            pos = random.sample(list(range(N)), 2)
            ind = self.childBoards[c][pos[1]]
            aux = np.delete(self.childBoards[c], pos[1])
            self.childBoards[c] = np.insert(aux, pos[0], ind)

        else: # default: exchange
            pos=random.sample(list(range(N)), 2) #Select two random positions
            self.childBoards[c][pos[1]], self.childBoards[c][pos[0]] = \
                self.childBoards[c][pos[0]], self.childBoards[c][pos[1]] #Swap positions

def main():

    if len(sys.argv) < 3:
        print("Usage: {} <population> <iteration> <mutation_rate>".format(sys.argv[0]))
        population = 10
        iterations = 10000 #Maximum number of iterations
        executions = 500 #Program executions to make average later
    else:
        population = sys.argv[1]
        iterations = sys.argv[2]
        executions = sys.argv[3]

    generationList=list()
    times=list()
    sample = GAQueen(population=population)

    #Best method configuration
    selectionMethod = "tournament"
    crossoverMethod = "order"
    mutationMethod = "inversion"

    for iter in range(executions): #GA executions

```

```
        start_time = time.time() #Reset counter time
        for parent in range(population):
            sample.initParent(parent) #Init population
        best=-N*(N-1)/2 # Maximum number of collisions

        p = 0 #Parent with best fitness, init.
        generation=0 #Count of iterations, init
        bestFromIteration=list() #Accumulate lists of best fitness for iteration, used for analysis purposes

        #Genetic algorithm main loop:
        while (best < 0) and generation<iterations : #best=0 means 0 collisions

            for childIndex in range(int(population/2)): #Methods loop
                parentIndex1=sample.selectWithReplacement(selectionMethod)
                parentIndex2=sample.selectWithReplacement(selectionMethod)
                sample.crossover(crossoverMethod, parentIndex1, parentIndex2, childIndex)
                sample.mutate(mutationMethod, childIndex) #Mutate first child
                sample.mutate(mutationMethod, childIndex+(int(population/2))) #Mutate second child

            sample.boards = sample.childBoards.copy()
            partialbests = -N * (N - 1) / 2 #Reset best fitness for this iteration

            for parent in range(population): #Fitness update loop
                sample.assessFitness(parent) #Calculate parent fitness

                if sample.fitness[parent] > partialbests:
                    partialbests=sample.fitness[parent] #Update best fitness for this iteration

                if sample.fitness[parent] > best:
                    best=sample.fitness[parent] #Update best total fitness
                    p = sample.boards[parent]

            bestFromIteration.append(partialbests)
            generation += 1

        generationList.append(generation) #Accumulate generation list for executions
        times.append(time.time() - start_time) #Accumulate time list for executions

    #Calculate average
    timeav=sum(times)/(float(len(times)))
    iterav = sum(generationList) / (float(len(times)))

    #Print stats
    print(timeav)
    print(iterav)
    print("Execution time: %s seconds" % (time.time() - start_time0)) #Total execution time
```

```
    print("\nPopulation:", population, " individuals")
    print("\nParent with best fitness:", p)
    generations=list(range(0,generation))
    plt.plot(generations, bestFromIteration) #Graphic of asymptotic convergence
    plt.xlabel("Generation")
    plt.ylabel("Best Fitness for every iteration")
    #plt.show()

if __name__ == '__main__':
    main()
```

A2: Python Code for 'N' Queens Solutions

```

import numpy as np
import os
import sys
import random
import matplotlib.pyplot as plt
import time
#import datetime
start_time = time.time()

N = 8 # for number of queens
xdata = []
ydata = []
plt.show()
axes = plt.gca()
axes.set_xlim(0, 1000)
axes.set_ylim(0, 50)
line, = axes.plot(xdata, ydata, 'r-')

csv_x_index = 0

if os.path.exists('time_evolution_Ngraph.csv'):
    os.remove('time_evolution_Ngraph.csv')

def draw_plot(data):
    if len(xdata) >= 1000:
        xdata.clear()
        ydata.clear()
    xdata.append(len(xdata))
    ydata.append(data)
    line.set_xdata(xdata)
    line.set_ydata(ydata)
    plt.xlabel("Queen Generation")
    plt.ylabel("Fitness(Cost) Value")
    plt.title("Fitness(Cost) Per Each Queen Generation")
    plt.draw()

```

```
    # write to csv file
    with open("time_evolution_Ngraph.csv", "a") as graph:
        global csv_x_index
        graph.write("{},{}\n".format(csv_x_index+1, data))
        csv_x_index += 1
        graph.close()

    plt.pause(1e-17)


class GAQueen:
    def __init__(self, population, iteration, mutation):
        self.population = population
        self.iteration = iteration
        self.mutation = mutation
        self.boardlength = N
        self.chromosome_matrix = np.zeros((30, 1000))
        self.cost_matrix = np.zeros(1000)
        self.crossovermatrix = np.zeros((30, 1000))
        self.area = np.zeros((30, 30))
        self.solutions = 0

    def clear(self):
        self.area = np.zeros((30, 30))

    def find_solution(self):
        positions = [-1] * self.boardlength
        self.put_queen(positions, 0)
        print("Number of total solutions: {}".format(self.solutions))
        print("Number of queens: {}".format(self.boardlength))

    def put_queen(self, positions, target_row):
        """
        Try to place a queen on target_row by checking all N possible cases.
        If a valid place is found the function calls itself trying to place a queen
        on the next row until all N queens are placed on the NxN board.
        """
        # Base (stop) case - all N rows are occupied
        if target_row == self.boardlength:
            # self.show_full_board(positions)
            self.show_short_board(positions)
            self.solutions += 1
        else:
            # For all N columns positions try to place a queen
            for column in range(self.boardlength):
                # Reject all invalid positions
                if self.check_place(positions, target_row, column):
                    positions[target_row] = column
                    self.put_queen(positions, target_row + 1)


    def check_place(self, positions, ocuppied_rows, column):
```

```
        """
        Check if a given position is under attack from any of
        the previously placed queens (check column and diagonal positions)
        """
        for i in range(ocuppied_rows):
            if positions[i] == column or positions[i] - i == column - ocuppied_rows or positions[i] + i == column + ocuppied_rows:
                return False
        return True

    def show_short_board(self, positions):
        """
        Show the queens positions on the board in compressed form,
        each number represent the occupied column position in the corresponding row.
        """
        line = "_".join(str(positions[i]) for i in range(self.boardlength))
        """
        print(line)
        """
        with open("final_Nsolution.csv", "a") as result:
            result.write(line + "\n")
            result.close()

    def cost_func(self, idx):
        cost_value = 0
        for i in range(self.boardlength):
            j = int(self.chromosome_matrix[i][idx])
            m = i + 1
            n = j - 1
            while m < self.boardlength and n >= 0:
                if int(self.area[m][n]) == 1:
                    cost_value += 1 # there is a queen that takes the other one
                m += 1
                n -= 1

            m = i + 1
            n = j + 1
            while m < self.boardlength and n < self.boardlength:
                if int(self.area[m][n]) == 1:
                    cost_value += 1
                m += 1
                n += 1

            m = i - 1
            n = j - 1
            while m >= 0 and n >= 0:
                if int(self.area[m][n]) == 1:
                    cost_value += 1
                m -= 1
                n -= 1

```

```
            m = i - 1
            n = j + 1
            while m >= 0 and n < self.boardlength:
                if int(self.area[m][n]) == 1:
                    cost_value += 1
                m -= 1
                n += 1

        return cost_value

    def initial_population(self):
        rand = 0
        check = False
        for index in range(self.population):
            a = 0
            while a < self.boardlength:
                rand = random.randrange(32768)
                check = 1
                for b in range(a):
                    if rand % self.boardlength ==
                        ↪ int(self.chromosome_matrix[b][index]):
                        check = 0
                if check:
                    self.chromosome_matrix[a][index] = rand % self.boardlength
                else:
                    a -= 1
                a += 1

    def population_sort(self):
        k = 1
        while k:
            k = 0
            for i in range(self.population-1):
                if int(self.cost_matrix[i]) > int(self.cost_matrix[i+1]):
                    temp = int(self.cost_matrix[i])
                    self.cost_matrix[i] = int(self.cost_matrix[i+1])
                    self.cost_matrix[i+1] = temp

                    for j in range(self.boardlength):
                        temp = int(self.chromosome_matrix[j][i])
                        self.chromosome_matrix[j][i] =
                            ↪ int(self.chromosome_matrix[j][i+1])
                        self.chromosome_matrix[j][i+1] = temp

                    k = 1

    def mating(self):
        temp_matrix = np.zeros((self.boardlength, 2))
        temp_matrix0 = np.zeros(self.boardlength)
        temp_matrix1 = np.zeros(self.boardlength)

        for index in range(self.population//4):
```

```
        for t in range(2):
            for i in range(self.boardlength):
                temp_matrix0[i] = int(self.chromosome_matrix[i][2 * index])
                temp_matrix1[i] = int(self.chromosome_matrix[i][2 * index + 1])

            for i in range(self.boardlength):
                if int(self.crossovermatrix[i][2*index+t]) == 0:
                    for j in range(self.boardlength):
                        if int(temp_matrix0[j]) != 100:
                            temp_matrix[i][t] = temp_matrix0[j]
                            temp = temp_matrix0[j]
                            temp_matrix0[j] = 100

                            for k in range(self.boardlength):
                                if int(temp_matrix1[k]) == temp:
                                    temp_matrix1[k] = 100
                                    break
                            break
                else:
                    for j in range(self.boardlength):
                        if int(temp_matrix1[j]) != 100:
                            temp_matrix[i][t] = temp_matrix1[j]
                            temp = temp_matrix1[j]
                            temp_matrix1[j] = 100

                            for k in range(self.boardlength):
                                if int(temp_matrix0[k]) == int(temp):
                                    temp_matrix0[k] = 100
                                    break
                            break

            for i in range(self.boardlength):
                self.chromosome_matrix[i][2*index+self.population//2+t] =
                    ↪ temp_matrix[i][t]

    def generate_crossovermatrix(self):
        for index in range(self.population):
            for a in range(self.boardlength):
                self.crossovermatrix[a][index] = random.randrange(32768) % 2

    def apply_mutation(self):
        number_mutation = int(self.mutation*(self.population-1)*self.boardlength)
        global rand_chromesome
        for k in range(number_mutation+1):
            rand_chromesome = 0
            while True:
                rand_chromesome = int(random.randrange(32768) % self.population)
                if rand_chromesome != 0:
                    break
            rand_gen0 = random.randrange(32768) % self.boardlength
            while True:
                rand_gen1 = random.randrange(32768) % self.boardlength
```

```
                if rand_gen1 != rand_gen0:
                    break

            temp = self.chromosome_matrix[rand_gen0][rand_chromesome]
            self.chromosome_matrix[rand_gen0][rand_chromesome] = self.chromosome_matrix[rand_gen1][rand_chromesome]
            self.chromosome_matrix[rand_gen0][rand_chromesome] = temp

    def fill_area(self, index):
        self.clear()
        for i in range(self.boardlength):
            self.area[i][int(self.chromosome_matrix[i][index])] = 1


def main():
    global mutation

    if len(sys.argv) < 4:
        print("Usage: {} <population> <iteration> <mutation_rate>".format(sys.argv[0]))
        population = 100
        iteration = 10
        mutation = 0.5
    else:
        population = sys.argv[1]
        iteration = sys.argv[2]
        mutation = sys.argv[3]

    sample = GAQueen(population=population, iteration=iteration, mutation=mutation)

    sample.initial_population()

    g = 0
    num = 0

    while g == 0 and num < sample.iteration:
        num += 1
        g = 0
        for k in range(sample.population):
            sample.fill_area(k)
            cost = sample.cost_func(k)
            sample.cost_matrix[k] = cost

            draw_plot(cost)

        sample.population_sort()

        if int(sample.cost_matrix[0]) == 0:
           g = 1

        sample.generate_crossovermatrix()
        sample.mating()
```

```
            sample.apply_mutation()

    res = '_'.join(str(int(sample.chromosome_matrix[i][0])) for i in range(N))
    print("Final result: {}".format(res))

    with open("final_Nsolution.csv", "w") as result:
        result.write("Final result: \n\t{}\n".format(res))
        result.write("\nAll solutions:\n")
        result.close()
    plt.show()

    sample.find_solution()
    with open("final_Nsolution.csv", "a") as result:
        result.write("\n\tTotal: {} solutions\n".format(sample.solutions))
        result.write("\n\tTotal number of queens: {}\n".format(N))
        result.write("\n\tDONE!!!\n")
        result.write("\n\tExecution time: %s seconds\n" % (time.time() - start_time))
        result.write("\n\tDate: {}". format(time.ctime()))
        result.close()
    print("DONE!!!")

if __name__ == '__main__':
    main()

    print("Execution time: %s seconds" % (time.time() - start_time))
    print(time.ctime())
```