

UNIVERSITAT AUTÒNOMA DE BARCELONA

---

# SOLVING 8-QUEENS PROBLEM WITH GENETIC ALGORITHMS: A SPECIAL INSTANCE PUZZLE

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

<b>Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 The problem: 8-queens problem . . . . .	3
<b>2 Definitions &amp; Concepts</b>	<b>4</b>
2.1 Art of the Game . . . . .	5
<b>3 Objectives</b>	<b>6</b>
<b>4 The Algorithm</b>	<b>7</b>
4.1 Ad-hoc Implementations . . . . .	7
4.2 Selection Methods . . . . .	8
4.3 Crossover Methods . . . . .	8
4.4 Mutation Methods . . . . .	9
<b>5 Code Implementation</b>	<b>11</b>
5.1 Class: GAQueen . . . . .	11
5.2 Variables: Structure of data . . . . .	11
5.3 Functions . . . . .	11
<b>6 Results</b>	<b>13</b>
6.1 Population . . . . .	13
6.2 Selection Methods . . . . .	15
6.3 Crossover Methods . . . . .	16
6.4 Mutation Methods . . . . .	16
6.5 Asymptotic convergence . . . . .	17
<b>7 Solutions</b>	<b>19</b>
7.1 Queens Solution Results . . . . .	19
7.2 Queens Solution Visualisation . . . . .	21
<b>8 Discussion and Conclusions</b>	<b>22</b>
8.1 Discussion . . . . .	22
8.2 Conclusion . . . . .	23
<b>References</b>	<b>24</b>
<b>Appendices</b>	<b>25</b>
<b>A1: Python Code for 'N' Queens Methods</b>	<b>25</b>
<b>A2: Python Code for 'N' Queens Solutions</b>	<b>30</b>

# 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 puzzle is the problem of placing eight chess queens on an  $8 \times 8$  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  $N \times N$  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 (**4x4**) 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**).

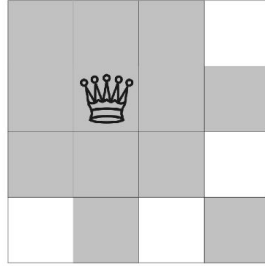


Figure 1: Intital Queen placement on (4x4) chess board from player 1

A (**8x8**) 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 (**4x4**) 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.

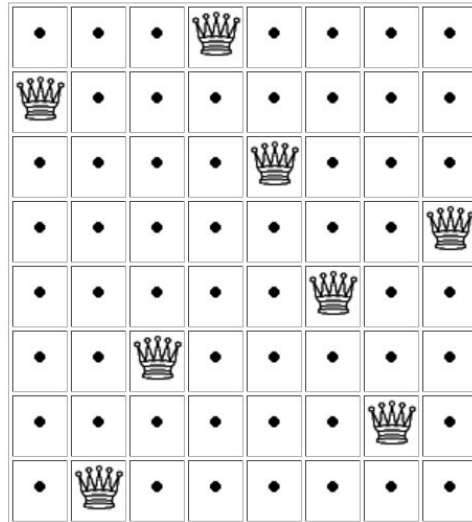


Figure 2: One solution to the 8-queens problem: 6 0 2 7 5 3 1 4

### 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  $N \times N$  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**.

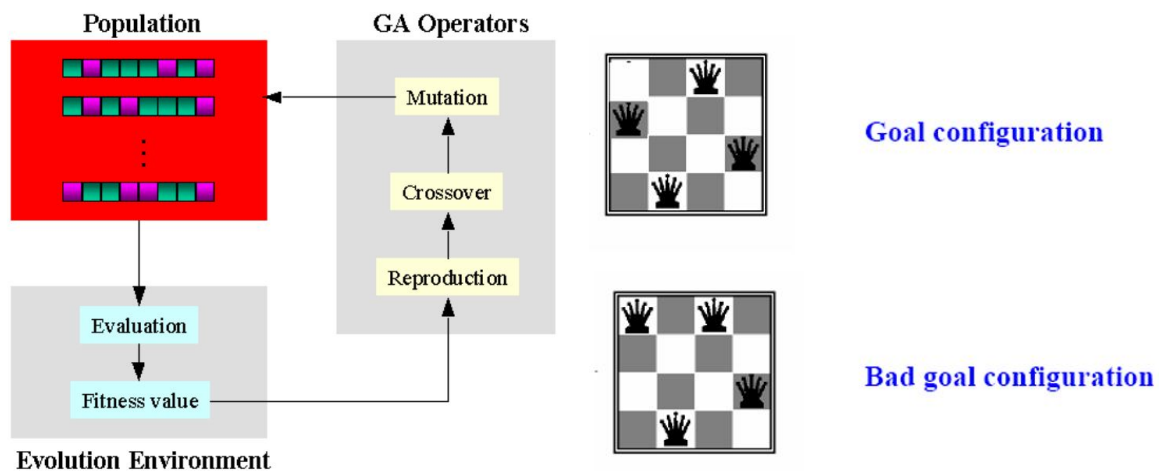


Figure 3: GA Evolution Flow with a "Good" vs "Bad" configuration solution

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:  $popsiz$   $\leftarrow$  desired population size ▷ This is basically  $\lambda$ . Make it even.
2:  $P \leftarrow \{\}$ 
3: for  $popsiz$  times do
4:    $P \leftarrow P \cup \{\text{new random individual}\}$ 
5:  $Best \leftarrow \square$ 
6: repeat
7:   for each individual  $P_i \in P$  do
8:     AssessFitness( $P_i$ )
9:     if  $Best = \square$  or Fitness( $P_i$ ) > Fitness( $Best$ ) then
10:       $Best \leftarrow P_i$ 
11:    $Q \leftarrow \{\}$  ▷ Here's where we begin to deviate from  $(\mu, \lambda)$ 
12:   for  $popsiz/2$  times do
13:     Parent  $P_a \leftarrow \text{SelectWithReplacement}(P)$ 
14:     Parent  $P_b \leftarrow \text{SelectWithReplacement}(P)$ 
15:     Children  $C_a, C_b \leftarrow \text{Crossover}(\text{Copy}(P_a), \text{Copy}(P_b))$ 
16:      $Q \leftarrow P \cup \{\text{Mutate}(C_a), \text{Mutate}(C_b)\}$ 
17:    $P \leftarrow Q$  ▷ 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 an 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  $N \times N$  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*

```

1: perform once per generation
2:   global  $\vec{p} \leftarrow$  population copied into a vector of individuals  $\langle p_1, p_2, \dots, p_l \rangle$ 

3:   global  $\vec{f} \leftarrow \langle f_1, f_2, \dots, f_l \rangle$  fitnesses of individuals in  $\vec{p}$  in the same order as  $\vec{p}$ 
4:   if  $\vec{f}$  is all 0.0s then ▷ Deal with all 0 fitnesses gracefully
5:     Convert  $\vec{f}$  to all 1.0s
6:   for  $i$  from 2 to  $l$  do ▷ Convert  $\vec{f}$  to a CDF. This will also cause  $f_l = s$ , the sum of fitnesses.
7:      $f_i \leftarrow f_i + f_{i-1}$ 
8:   perform each time
9:      $n \leftarrow$  random number from 0 to  $f_l$  inclusive
10:    for  $i$  from 2 to  $l$  do ▷ This could be done more efficiently with binary search
11:      if  $f_{i-1} < n \leq f_i$  then
12:        return  $p_i$ 
13:  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*

- 1:  $P \leftarrow$  population
- 2:  $t \leftarrow$  tournament size,  $t \geq 1$
- 3:  $Best \leftarrow$  individual picked at random from  $P$  with replacement
- 4: **for**  $i$  from 2 to  $t$  **do**
- 5:    $Next \leftarrow$  individual picked at random from  $P$  with replacement
- 6:   **if**  $Fitness(Next) > Fitness(Best)$  **then**
- 7:      $Best \leftarrow Next$
- 8: **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 2<sup>nd</sup> parent. The resulted sequence of cities 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.

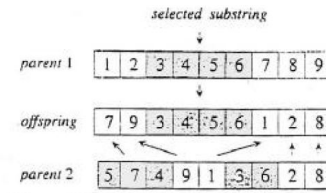


Figure 7: Illustration of OX Operator

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

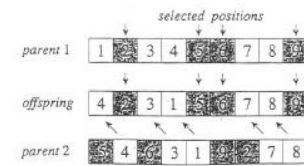


Figure 8: Illustration of Position-Based Operator

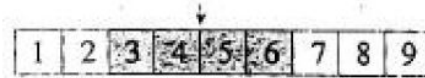
## 4.4 Mutation Methods

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.

***Inversion Mutation :***

*select a subtour at random*

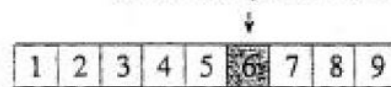


*invert the substring*



***Insertion Mutation :***

*select a city at random*

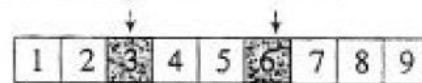


*insert it in a random position*



***Exchange Mutation :***

*select two positions at random*



*swap the relative cities*

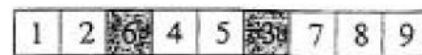


Figure 9: Illustration of inversion, insertion and exchange mutation respectively

## 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.

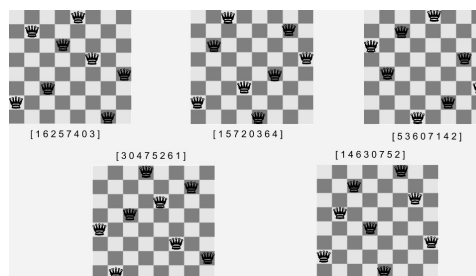


Figure 10: Genetic Algorithms Solutions and Chessboard Locations

## 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**

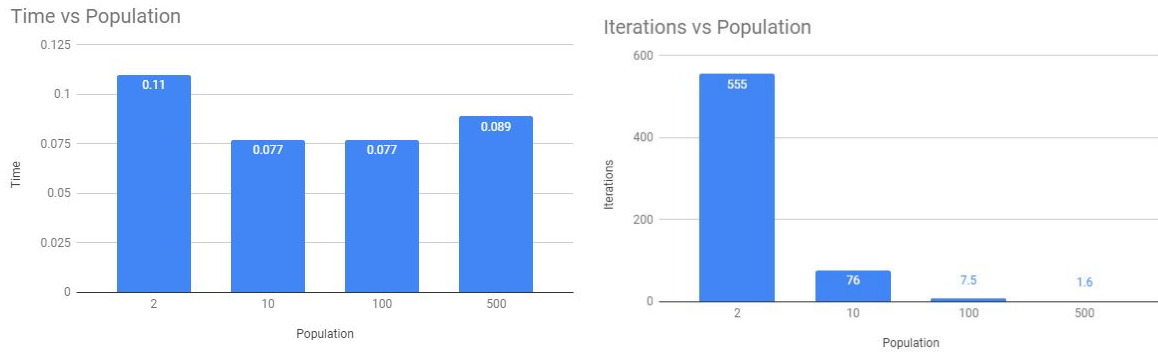


Figure 11: Graphics of execution time, iterations and population

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:

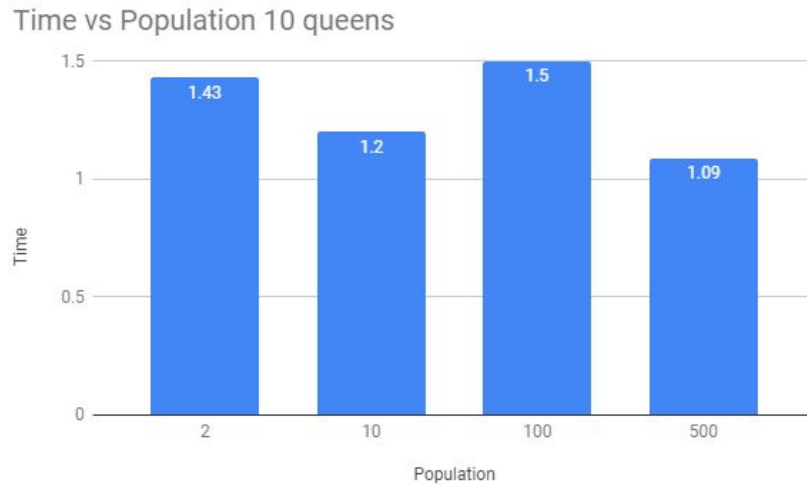


Figure 12: Graphics of execution time and population for 10 queens

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.

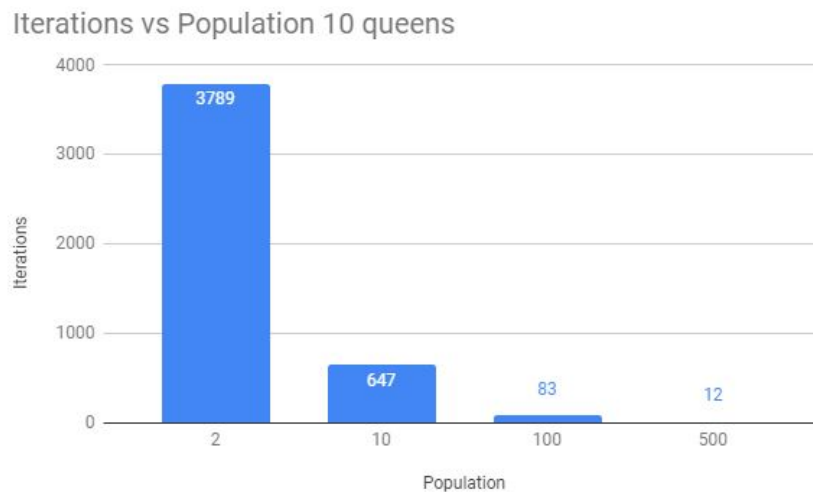


Figure 13: Graphics of iterations and population for 10 queens

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:

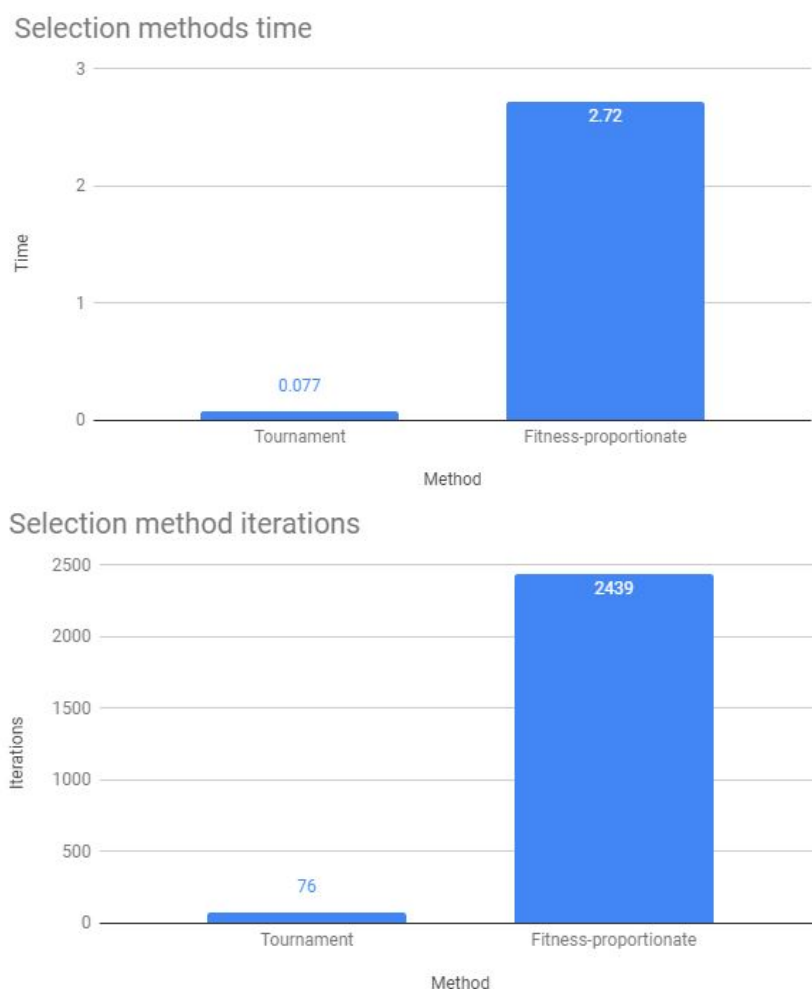


Figure 14: Graphics of time and iterations for selection methods

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:

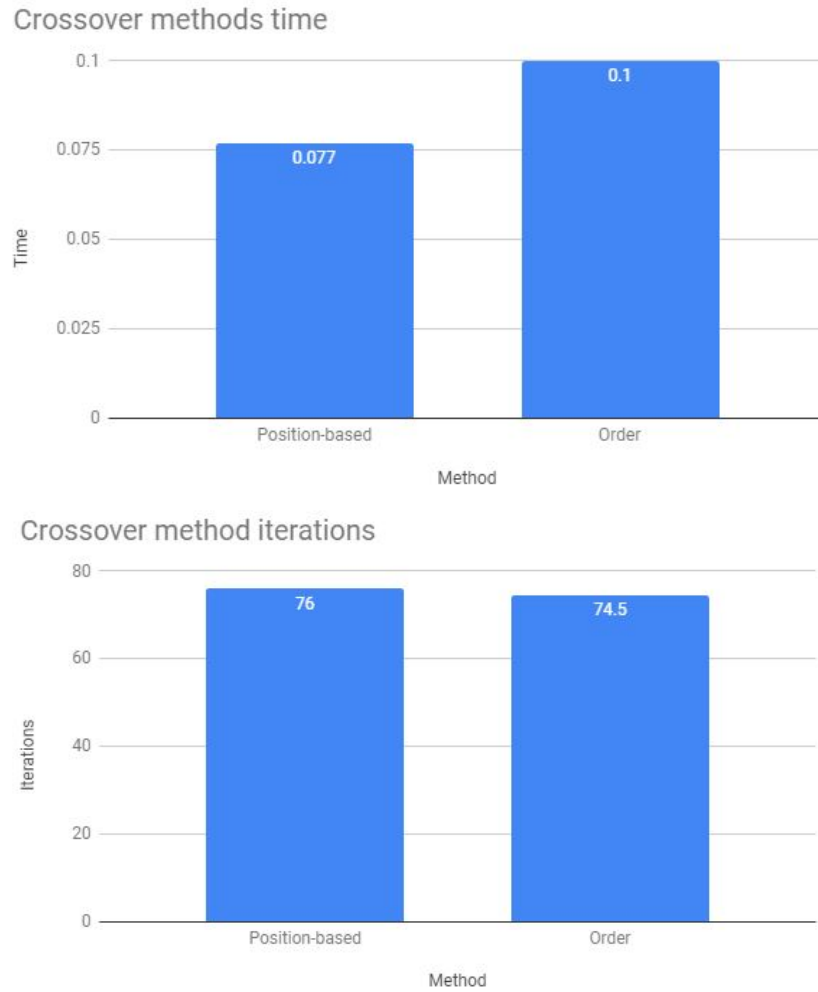


Figure 15: Graphics of time and iterations for crossover methods

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:

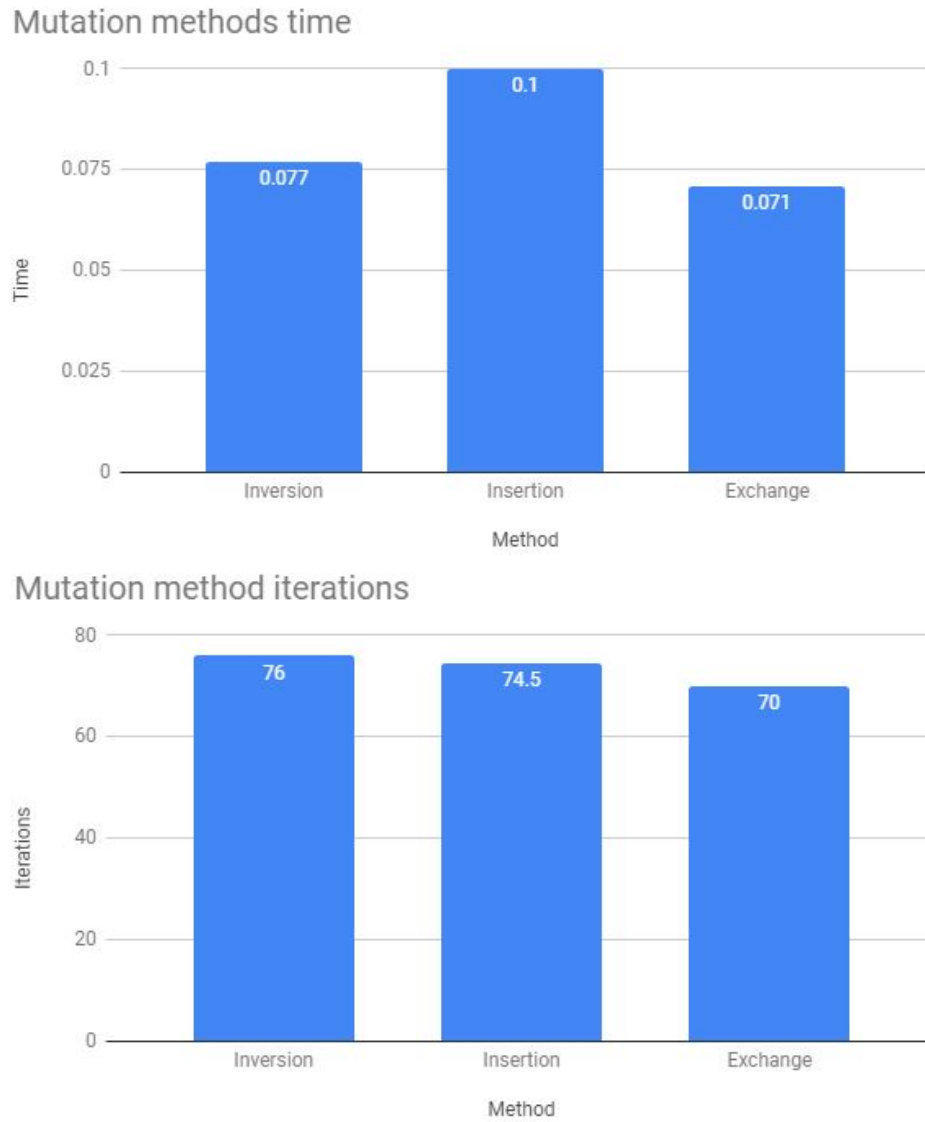


Figure 16: Graphics of time and iterations for mutation methods

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:

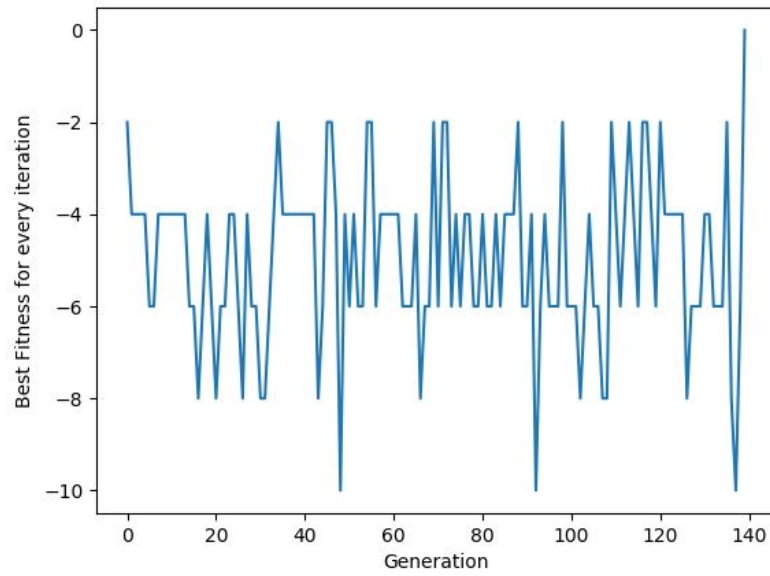


Figure 17: Graphic of asymptotic convergence for 8 queens, 10 population size and best methods combination

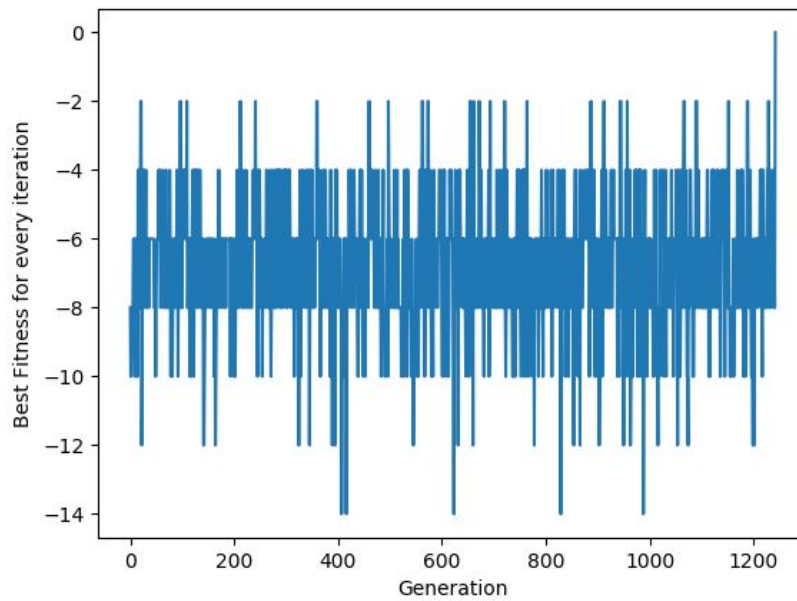


Figure 18: Graphic of asymptotic convergence for 10 queens, 10 population size and best methods combination

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  $N = 8$ ):

---

```

1
2 Final result:
3 6_0_2_7_5_3_1_4
4
5 All solutions:
6 0_4_7_5_2_6_1_3
7 0_5_7_2_6_3_1_4
8 0_6_3_5_7_1_4_2
9 0_6_4_7_1_3_5_2
10 1_3_5_7_2_0_6_4
11 1_4_6_0_2_7_5_3
12 1_4_6_3_0_7_5_2
13 1_5_0_6_3_7_2_4
14 1_5_7_2_0_3_6_4
15 1_6_2_5_7_4_0_3
16 1_6_4_7_0_3_5_2
17 1_7_5_0_2_4_6_3
18 2_0_6_4_7_1_3_5
19 2_4_1_7_0_6_3_5
20 2_4_1_7_5_3_6_0
21 2_4_6_0_3_1_7_5
22 2_4_7_3_0_6_1_5
23 2_5_1_4_7_0_6_3
24 2_5_1_6_0_3_7_4
25 2_5_1_6_4_0_7_3
26 2_5_3_0_7_4_6_1
27 2_5_3_1_7_4_6_0
28 2_5_7_0_3_6_4_1
29 2_5_7_0_4_6_1_3
30 2_5_7_1_3_0_6_4
31 2_6_1_7_4_0_3_5
32 2_6_1_7_5_3_0_4
33 2_7_3_6_0_5_1_4
34 3_0_4_7_1_6_2_5
35 3_0_4_7_5_2_6_1
36 3_1_4_7_5_0_2_6
37 3_1_6_2_5_7_0_4
38 3_1_6_2_5_7_4_0
39 3_1_6_4_0_7_5_2
40 3_1_7_4_6_0_2_5

```

---

41 3\_1\_7\_5\_0\_2\_4\_6  
42 3\_5\_0\_4\_1\_7\_2\_6  
43 3\_5\_7\_1\_6\_0\_2\_4  
44 3\_5\_7\_2\_0\_6\_4\_1  
45 3\_6\_0\_7\_4\_1\_5\_2  
46 3\_6\_2\_7\_1\_4\_0\_5  
47 3\_6\_4\_1\_5\_0\_2\_7  
48 3\_6\_4\_2\_0\_5\_7\_1  
49 3\_7\_0\_2\_5\_1\_6\_4  
50 3\_7\_0\_4\_6\_1\_5\_2  
51 3\_7\_4\_2\_0\_6\_1\_5  
52 4\_0\_3\_5\_7\_1\_6\_2  
53 4\_0\_7\_3\_1\_6\_2\_5  
54 4\_0\_7\_5\_2\_6\_1\_3  
55 4\_1\_3\_5\_7\_2\_0\_6  
56 4\_1\_3\_6\_2\_7\_5\_0  
57 4\_1\_5\_0\_6\_3\_7\_2  
58 4\_1\_7\_0\_3\_6\_2\_5  
59 4\_2\_0\_5\_7\_1\_3\_6  
60 4\_2\_0\_6\_1\_7\_5\_3  
61 4\_2\_7\_3\_6\_0\_5\_1  
62 4\_6\_0\_2\_7\_5\_3\_1  
63 4\_6\_0\_3\_1\_7\_5\_2  
64 4\_6\_1\_3\_7\_0\_2\_5  
65 4\_6\_1\_5\_2\_0\_3\_7  
66 4\_6\_1\_5\_2\_0\_7\_3  
67 4\_6\_3\_0\_2\_7\_5\_1  
68 4\_7\_3\_0\_2\_5\_1\_6  
69 4\_7\_3\_0\_6\_1\_5\_2  
70 5\_0\_4\_1\_7\_2\_6\_3  
71 5\_1\_6\_0\_2\_4\_7\_3  
72 5\_1\_6\_0\_3\_7\_4\_2  
73 5\_2\_0\_6\_4\_7\_1\_3  
74 5\_2\_0\_7\_3\_1\_6\_4  
75 5\_2\_0\_7\_4\_1\_3\_6  
76 5\_2\_4\_6\_0\_3\_1\_7  
77 5\_2\_4\_7\_0\_3\_1\_6  
78 5\_2\_6\_1\_3\_7\_0\_4  
79 5\_2\_6\_1\_7\_4\_0\_3  
80 5\_2\_6\_3\_0\_7\_1\_4  
81 5\_3\_0\_4\_7\_1\_6\_2  
82 5\_3\_1\_7\_4\_6\_0\_2  
83 5\_3\_6\_0\_2\_4\_1\_7  
84 5\_3\_6\_0\_7\_1\_4\_2  
85 5\_7\_1\_3\_0\_6\_4\_2  
86 6\_0\_2\_7\_5\_3\_1\_4  
87 6\_1\_3\_0\_7\_4\_2\_5  
88 6\_1\_5\_2\_0\_3\_7\_4  
89 6\_2\_0\_5\_7\_4\_1\_3  
90 6\_2\_7\_1\_4\_0\_5\_3  
91 6\_3\_1\_4\_7\_0\_2\_5  
92 6\_3\_1\_7\_5\_0\_2\_4

```

93 6_4_2_0_5_7_1_3
94 7_1_3_0_6_4_2_5
95 7_1_4_2_0_6_3_5
96 7_2_0_5_1_4_6_3
97 7_3_0_2_5_1_6_4
98
99 Total: 92 solutions
100
101 Total number of queens: 8
102
103 DONE!!!
104
105 Execution time: 6.477712631225586 seconds
106
107 Date: Sat Nov 24 20:55:39 2018

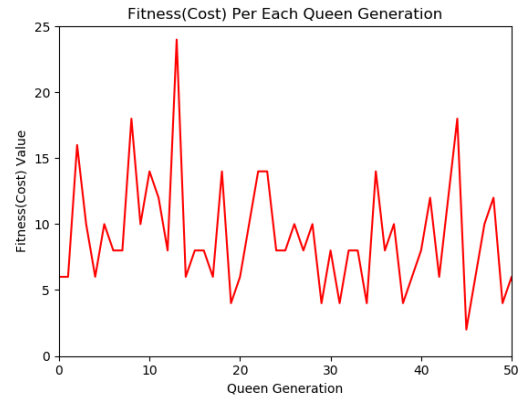
```

## 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.

Final Result: 6\_ 0\_ 2\_ 7\_ 5\_ 3\_ 1\_ 4

7				Q				
6	Q							
5					Q			
4							Q	
3					Q			
2			Q					
1							Q	
0		Q						
	0	1	2	3	4	5	6	7



No. of Queens	One Solution (with Python Code)	Execution Time (seconds)	Solution Set Size (n!)	No. of Solutions
1	0	0.0000000000000000	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  $N \times N$  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.

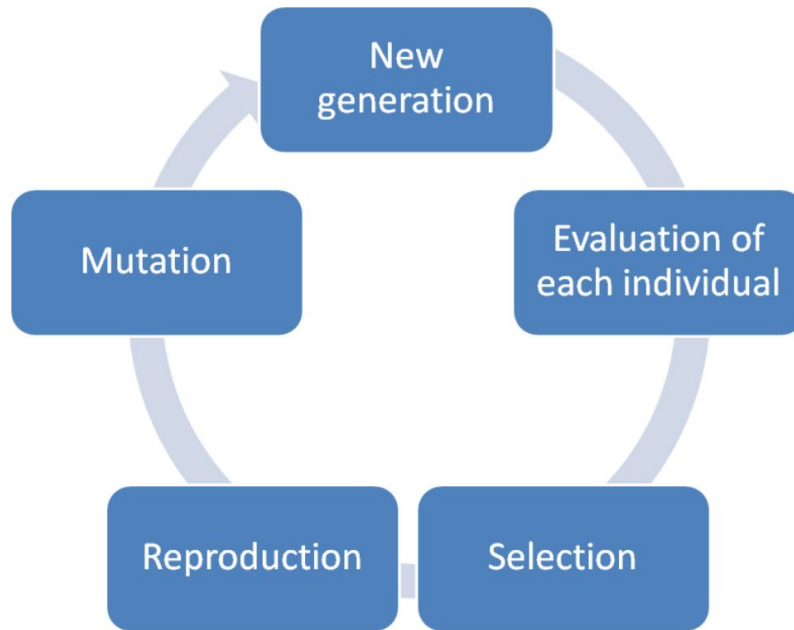


Figure 20: A Genetic Algorithm process over continuous sequential generations

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 performing 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. McCulloch. 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](https://www.tutorialspoint.com/genetic_algorithms).
- [7] L. Alseda. 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](http://www.durangobill.com/N_Queens.html).

# Appendices

## A1: Python Code for 'N' Queens Methods

---

```

1
2 import numpy as np
3 import sys
4 import random
5 import matplotlib.pyplot as plt
6 import time
7 import math
8
9 """
10 Ad-hoc supposition: Since queens can't collide between rows and columns, we can
11   ↪ reinterpret the queen positions on board
12   as an array, where index of the array is the board row and the values are the
13   ↪ column positions.
14   Then, we only need to obtain the correct arrangements (solutions of N queens)
15   """
16
17 start_time0 = time.time() # Init global time counter
18 N = 8 # Number of queens
19
20 class GAQueen:
21     def __init__(self, population):
22         self.population = population
23         self.boards = np.zeros((population, N))
24         self.childBoards = np.zeros((population, N))
25         self.fitness = np.empty(population)
26         self.fitness.fill(-N * (N - 1) / 2) # Maximum number of collisions
27
28     def initParent(self, a):
29         """
30         Assign range(N) elements randomly ordered to parent and child a
31         :param a: Index of parent
32         """
33
34         numbers=list(range(N)) #Generate N numbers from 0 to N-1 (the N values for
35         ↪ N positions)
36         random.shuffle(numbers) #Reorder numbers list randomly
37         self.boards[a,:]=numbers #Assign to board
38         self.childBoards[a, :] = numbers #Assign to children
39
40     def assessFitness(self, a):
41         """
42         Check the number of collisions between queens. We only have to check
43         ↪ diagonal collisions
44         since queens are in different rows and columns every time. Less
45         ↪ collisions means best
46         fitness, with maximum fitness of 0

```

```

42         :param a: Index of parent
43         """
44
45         collisions=0 #Reset collisions
46         board= self.boards[a,:] # Get board parent
47
48         # check diagonal collisions
49         for i in range(N):
50             for j in range(N):
51                 if(i!=j):
52                     deltaRow= abs(i - j)
53                     deltaCol= abs(board[i]-board[j])
54                     if deltaRow == deltaCol: collisions+=1
55
56         self.fitness[a]=collisions*(-1)
57
58     def selectWithReplacement(self,method):
59         """
60         Returns Parent in population according to the chosen method
61         :param method: Method selected
62         :return: Index of parent selected
63         """
64
65         if method == 'fitness-proportionate':
66             fitness = self.fitness.copy()
67             n = random.randrange(0, self.population)
68             best=-1
69             for i in range (1,self.population):
70                 fitness[i]=self.fitness[i]+self.fitness[i-1] #CDF of fitnesses.
71                 ↪ Fitness[population] = the sum of fitnesses
72                 if fitness[i-1] < n <= fitness[i]:
73                     best=i
74             if best==-1 : best=self.population-1 #If best was not modified, it will
75                 ↪ be last index
76
77         elif method == 'tournament':
78             t=2 #Tournament size, most popular size is 2
79             best = random.randrange(0, self.population)
80             for j in range (1,t):
81                 next = random.randrange(0, self.population)
82                 if self.fitness[next] > self.fitness[best]:
83                     best = next
84
85         else: # default: Complete random selection, no selection by fitness
86             best = random.randrange(0, self.population)
87
88         return best
89
90     def crossover(self,method,a,b,c):
91         """
92         Crossover two parents of boards population and update childBoards
93         ↪ according to the chosen method

```

```

91         :param method: Method selected
92         :param a: Index of first parent
93         :param b: Index of second parent
94         :param c: Index of first child
95     """
96
97     d = c + int(self.population/2) #Index of second child
98     self.childBoards[c] = self.boards[a] # Copy parent to child
99     self.childBoards[d] = self.boards[b] # Copy parent to child
100
101     if method == 'position-based':
102         #Note order-based is the inverse of position-based, and it's done also
103         ↪ in this method to get the other child
104
105         for z in range(0, 2): # For the two children
106             fixedValues = random.sample(list(range(N)), int(N / 2)) # Fixed
107             ↪ values
108             nonFixedList = list() # Init "long" substring (in case N uneven)
109             ↪ from parent2
110             for j in range(0, N):
111                 if self.boards[b][j] not in fixedValues:
112                     nonFixedList.append(self.boards[b][j]) # Get non fixed
113                     ↪ elements from parent2
114             for j in range(0, N):
115                 if self.childBoards[c][j] not in fixedValues:
116                     self.childBoards[c][j] = nonFixedList.pop(0) # Get non fixed
117                     ↪ elements from parent2
118             b, a, c = a, b, d # Swap index to generate the other child
119
120     else: # default: order
121         for z in range(0, 2):
122
123             start = random.randrange(0, math.ceil(N / 2)) # start of fixed
124             ↪ substring (index included)
125             end = start + int(N / 2) # End of fixed substring (index not
126             ↪ included)
127
128             nonFixedList = list() # Init "long" substring (in case N uneven)
129             ↪ from parent2
130             for j in range(0, N):
131                 # If element in b is not in fixed substring [from start through
132                 ↪ end-1, where maximum end is equal to N]
133                 if self.boards[b][j] not in self.boards[a][start:end]:
134                     nonFixedList.append(self.boards[b][j]) # Get non fixed
135                     ↪ elements from parent2
136
137             # Assign non fixed elements to childBoards "holes"
138             for i in range(0, start): self.childBoards[c][i] =
139                 ↪ nonFixedList.pop(0)
140             for i in range(end, N): self.childBoards[c][i] = nonFixedList.pop(0)
141             b, a, c = a, b, d # Swap index to generate the other child

```

```

132 def mutate(self,method,c):
133     """
134         Mutate a childBoards according to the chosen method
135         :param method: Method selected
136         :param c: Index of one childBoards
137     """
138     if method == 'inversion':
139         start = random.randrange(0, math.ceil(N / 2)) # start of fixed
140             ↪ substring (index included)
141         end = start + int(N / 2) # End of fixed substring (index not included)
142         substringReversed = self.childBoards[c][start:end][::-1].copy()
143         j = 0
144         for i in range(start, end):
145             self.childBoards[c][i] = substringReversed[j]
146             j += 1
147
148     elif method == 'insertion':
149         pos = random.sample(list(range(N)), 2)
150         ind = self.childBoards[c][pos[1]]
151         aux = np.delete(self.childBoards[c], pos[1])
152         self.childBoards[c] = np.insert(aux, pos[0], ind)
153
154     else: # default: exchange
155         pos=random.sample(list(range(N)), 2) #Select two random positions
156         self.childBoards[c][pos[1]], self.childBoards[c][pos[0]] = \
157             self.childBoards[c][pos[0]], self.childBoards[c][pos[1]] #Swap
158             ↪ positions
159
160 def main():
161     if len(sys.argv) < 3:
162         print("Usage: {} {}<population> {}<iteration> {}  

163             ↪ <mutation_rate>".format(sys.argv[0]))
164         population = 10
165         iterations = 10000 #Maximum number of iterations
166         executions = 500 #Program executions to make average later
167     else:
168         population = sys.argv[1]
169         iterations = sys.argv[2]
170         executions = sys.argv[3]
171
172     generationList=list()
173     times=list()
174     sample = GAQueen(population=population)
175
176     #Best method configuration
177     selectionMethod = "tournament"
178     crossoverMethod = "order"
179     mutationMethod = "inversion"
180
181     for iter in range(executions): #GA executions

```

```

181     start_time = time.time() #Reset counter time
182     for parent in range(population):
183         sample.initParent(parent) #Init population
184     best=-N*(N-1)/2 # Maximum number of collisions
185
186     p = 0 #Parent with best fitness, init.
187     generation=0 #Count of iterations, init
188     bestFromIteration=list() #Accumulate lists of best fitness for iteration,
        ↪ used for analysis purposes
189
190     #Genetic algorithm main loop:
191     while (best < 0) and generation<iterations : #best=0 means 0 collisions
192
193         for childIndex in range(int(population/2)): #Methods loop
194             parentIndex1=sample.selectWithReplacement(selectionMethod)
195             parentIndex2=sample.selectWithReplacement(selectionMethod)
196             sample.crossover(crossoverMethod, parentIndex1, parentIndex2,
        ↪ childIndex)
197             sample.mutate(mutationMethod, childIndex) #Mutate first child
198             sample.mutate(mutationMethod, childIndex+(int(population/2)))
        ↪ #Mutate second child
199
200         sample.boards = sample.childBoards.copy()
201         partialbests = -N * (N - 1) / 2 #Reset best fitness for this iteration
202
203         for parent in range(population): #Fitness update loop
204             sample.assessFitness(parent) #Calculate parent fitness
205
206             if sample.fitness[parent] > partialbests:
207                 partialbests=sample.fitness[parent] #Update best fitness for
        ↪ this iteration
208
209             if sample.fitness[parent] > best:
210                 best=sample.fitness[parent] #Update best total fitness
211                 p = sample.boards[parent]
212
213         bestFromIteration.append(partialbests)
214         generation += 1
215
216     generationList.append(generation) #Accumulate generation list for
        ↪ executions
217     times.append(time.time() - start_time) #Accumulate time list for executions
218
219     #Calculate average
220     timeav=sum(times)/(float(len(times)))
221     iterav = sum(generationList) / (float(len(times)))
222
223     #Print stats
224     print(timeav)
225     print(iterav)
226     print("Execution_␣time:␣%s␣seconds" % (time.time() - start_time0)) #Total
        ↪ execution time

```

---

```

227     print("\nPopulation:", population, "\nindividuals")
228     print("\nParent with best fitness:", p)
229     generations=list(range(0,generation))
230     plt.plot(generations, bestFromIteration) #Graphic of asymptotic convergence
231     plt.xlabel("Generation")
232     plt.ylabel("Best Fitness for every iteration")
233     #plt.show()
234
235 if __name__ == '__main__':
236     main()

```

---

## A2: Python Code for 'N' Queens Solutions

---

```

1
2 import numpy as np
3 import os
4 import sys
5 import random
6 import matplotlib.pyplot as plt
7 import time
8 #import datetime
9 start_time = time.time()
10
11 N = 8 # for number of queens
12 xdata = []
13 ydata = []
14 plt.show()
15 axes = plt.gca()
16 axes.set_xlim(0, 1000)
17 axes.set_ylim(0, 50)
18 line, = axes.plot(xdata, ydata, 'r-')
19
20 csv_x_index = 0
21
22 if os.path.exists('time_evolution_Ngraph.csv'):
23     os.remove('time_evolution_Ngraph.csv')
24
25 def draw_plot(data):
26     if len(xdata) >= 1000:
27         xdata.clear()
28         ydata.clear()
29     xdata.append(len(xdata))
30     ydata.append(data)
31     line.set_xdata(xdata)
32     line.set_ydata(ydata)
33     plt.xlabel("Queen Generation")
34     plt.ylabel("Fitness(Cost) Value")
35     plt.title("Fitness(Cost) Per Each Queen Generation")
36     plt.draw()
37

```

```

38     # write to csv file
39     with open("time_evolution_Ngraph.csv", "a") as graph:
40         global csv_x_index
41         graph.write("{}{}\n".format(csv_x_index+1, data))
42         csv_x_index += 1
43         graph.close()
44
45     plt.pause(1e-17)
46
47
48     class GAQueen:
49         def __init__(self, population, iteration, mutation):
50             self.population = population
51             self.iteration = iteration
52             self.mutation = mutation
53             self.boardlength = N
54             self.chromosome_matrix = np.zeros((30, 1000))
55             self.cost_matrix = np.zeros(1000)
56             self.crossovermatrix = np.zeros((30, 1000))
57             self.area = np.zeros((30, 30))
58             self.solutions = 0
59
60         def clear(self):
61             self.area = np.zeros((30, 30))
62
63         def find_solution(self):
64             positions = [-1] * self.boardlength
65             self.put_queen(positions, 0)
66             print("Number_of_total_solutions: {}".format(self.solutions))
67             print("Number_of_queens: {}".format(self.boardlength))
68
69         def put_queen(self, positions, target_row):
70             """
71             Try to place a queen on target_row by checking all N possible cases.
72             If a valid place is found the function calls itself trying to place a queen
73             on the next row until all N queens are placed on the NxN board.
74             """
75             # Base (stop) case - all N rows are occupied
76             if target_row == self.boardlength:
77                 # self.show_full_board(positions)
78                 self.show_short_board(positions)
79                 self.solutions += 1
80             else:
81                 # For all N columns positions try to place a queen
82                 for column in range(self.boardlength):
83                     # Reject all invalid positions
84                     if self.check_place(positions, target_row, column):
85                         positions[target_row] = column
86                         self.put_queen(positions, target_row + 1)
87
88
89         def check_place(self, positions, occupied_rows, column):

```



```

90     """
91     Check if a given position is under attack from any of
92     the previously placed queens (check column and diagonal positions)
93     """
94     for i in range(occupied_rows):
95         if positions[i] == column or positions[i] - i == column - occupied_rows
96             ↪ or positions[i] + i == column + occupied_rows:
97             return False
98     return True
99
100 def show_short_board(self, positions):
101     """
102     Show the queens positions on the board in compressed form,
103     each number represent the occupied column position in the corresponding
104     ↪ row.
105     """
106     line = "_".join(str(positions[i]) for i in range(self.boardlength))
107     print(line)
108     with open("final_Nsolution.csv", "a") as result:
109         result.write(line + "\n")
110         result.close()
111
112 def cost_func(self, idx):
113     cost_value = 0
114     for i in range(self.boardlength):
115         j = int(self.chromosome_matrix[i][idx])
116         m = i + 1
117         n = j - 1
118         while m < self.boardlength and n >= 0:
119             if int(self.area[m][n]) == 1:
120                 cost_value += 1 # there is a queen that takes the other one
121                 m += 1
122                 n -= 1
123
124         m = i + 1
125         n = j + 1
126         while m < self.boardlength and n < self.boardlength:
127             if int(self.area[m][n]) == 1:
128                 cost_value += 1
129                 m += 1
130                 n += 1
131
132         m = i - 1
133         n = j - 1
134         while m >= 0 and n >= 0:
135             if int(self.area[m][n]) == 1:
136                 cost_value += 1
137                 m -= 1
138                 n -= 1
139

```

```

140         m = i - 1
141         n = j + 1
142         while m >= 0 and n < self.boardlength:
143             if int(self.area[m][n]) == 1:
144                 cost_value += 1
145             m -= 1
146             n += 1
147
148         return cost_value
149
150     def initial_population(self):
151         rand = 0
152         check = False
153         for index in range(self.population):
154             a = 0
155             while a < self.boardlength:
156                 rand = random.randrange(32768)
157                 check = 1
158                 for b in range(a):
159                     if rand % self.boardlength ==
160                         ↪ int(self.chromosome_matrix[b][index]):
161                         check = 0
162                 if check:
163                     self.chromosome_matrix[a][index] = rand % self.boardlength
164                 else:
165                     a -= 1
166                 a += 1
167
168     def population_sort(self):
169         k = 1
170         while k:
171             k = 0
172             for i in range(self.population-1):
173                 if int(self.cost_matrix[i]) > int(self.cost_matrix[i+1]):
174                     temp = int(self.cost_matrix[i])
175                     self.cost_matrix[i] = int(self.cost_matrix[i+1])
176                     self.cost_matrix[i+1] = temp
177
178             for j in range(self.boardlength):
179                 temp = int(self.chromosome_matrix[j][i])
180                 self.chromosome_matrix[j][i] =
181                     ↪ int(self.chromosome_matrix[j][i+1])
182                 self.chromosome_matrix[j][i+1] = temp
183
184             k = 1
185
186     def mating(self):
187         temp_matrix = np.zeros((self.boardlength, 2))
188         temp_matrix0 = np.zeros(self.boardlength)
189         temp_matrix1 = np.zeros(self.boardlength)
190
191         for index in range(self.population//4):

```

```

190         for t in range(2):
191             for i in range(self.boardlength):
192                 temp_matrix0[i] = int(self.chromosome_matrix[i][2 * index])
193                 temp_matrix1[i] = int(self.chromosome_matrix[i][2 * index + 1])
194
195             for i in range(self.boardlength):
196                 if int(self.crossovermatrix[i][2*index+t]) == 0:
197                     for j in range(self.boardlength):
198                         if int(temp_matrix0[j]) != 100:
199                             temp_matrix[i][t] = temp_matrix0[j]
200                             temp = temp_matrix0[j]
201                             temp_matrix0[j] = 100
202
203                         for k in range(self.boardlength):
204                             if int(temp_matrix1[k]) == temp:
205                                 temp_matrix1[k] = 100
206                                 break
207                     break
208             else:
209                 for j in range(self.boardlength):
210                     if int(temp_matrix1[j]) != 100:
211                         temp_matrix[i][t] = temp_matrix1[j]
212                         temp = temp_matrix1[j]
213                         temp_matrix1[j] = 100
214
215                     for k in range(self.boardlength):
216                         if int(temp_matrix0[k]) == int(temp):
217                             temp_matrix0[k] = 100
218                             break
219                     break
220
221             for i in range(self.boardlength):
222                 self.chromosome_matrix[i][2*index+self.population//2+t] =
223                 ↪ temp_matrix[i][t]
224
225     def generate_crossovermatrix(self):
226         for index in range(self.population):
227             for a in range(self.boardlength):
228                 self.crossovermatrix[a][index] = random.randrange(32768) % 2
229
230     def apply_mutation(self):
231         number_mutation = int(self.mutation*(self.population-1)*self.boardlength)
232         global rand_chromosome
233         for k in range(number_mutation+1):
234             rand_chromosome = 0
235             while True:
236                 rand_chromosome = int(random.randrange(32768) % self.population)
237                 if rand_chromosome != 0:
238                     break
239             rand_gen0 = random.randrange(32768) % self.boardlength
240             while True:
241                 rand_gen1 = random.randrange(32768) % self.boardlength

```

```

241         if rand_gen1 != rand_gen0:
242             break
243
244         temp = self.chromosome_matrix[rand_gen0][rand_chromosome]
245         self.chromosome_matrix[rand_gen0][rand_chromosome] =
246             ↪ self.chromosome_matrix[rand_gen1][rand_chromosome]
247         self.chromosome_matrix[rand_gen0][rand_chromosome] = temp
248
249     def fill_area(self, index):
250         self.clear()
251         for i in range(self.boardlength):
252             self.area[i][int(self.chromosome_matrix[i][index])] = 1
253
254     def main():
255         global mutation
256
257         if len(sys.argv) < 4:
258             print("Usage: {} {}<population> {}<iteration> {}
259                 ↪ <mutation_rate>".format(sys.argv[0]))
260             population = 100
261             iteration = 10
262             mutation = 0.5
263         else:
264             population = sys.argv[1]
265             iteration = sys.argv[2]
266             mutation = sys.argv[3]
267
268         sample = GAQueen(population=population, iteration=iteration, mutation=mutation)
269
270         sample.initial_population()
271
272         g = 0
273         num = 0
274
275         while g == 0 and num < sample.iteration:
276             num += 1
277             g = 0
278             for k in range(sample.population):
279                 sample.fill_area(k)
280                 cost = sample.cost_func(k)
281                 sample.cost_matrix[k] = cost
282
283                 draw_plot(cost)
284
285             sample.population_sort()
286
287             if int(sample.cost_matrix[0]) == 0:
288                 g = 1
289
290             sample.generate_crossovermatrix()
291             sample.mating()

```

```
291         sample.apply_mutation()
292
293     res = '_'.join(str(int(sample.chromosome_matrix[i][0])) for i in range(N))
294     print("Final_result: {}".format(res))
295
296     with open("final_Nsolution.csv", "w") as result:
297         result.write("Final_result: {}\n\t{}\n".format(res))
298         result.write("\nAll_solutions:\n")
299         result.close()
300     plt.show()
301
302     sample.find_solution()
303     with open("final_Nsolution.csv", "a") as result:
304         result.write("\n\tTotal: {}_solutions\n".format(sample.solutions))
305         result.write("\n\tTotal_number_of_queens: {}\n".format(N))
306         result.write("\n\tDONE!!!\n")
307         result.write("\n\tExecution_time: {}s_seconds\n" % (time.time() -
308             ↪ start_time))
309         result.write("\n\tDate: {}".format(time.ctime()))
310         result.close()
311     print("DONE!!!")
312
313 if __name__ == '__main__':
314     main()
315
316     print("Execution_time: {}s_seconds" % (time.time() - start_time))
317     print(time.ctime())
```

---