Solving the 8 queens problem using genetic algorithms

Alejandro Encalado Masia and Albert Xavier Lopez Barrantes

24 de noviembre de 2018

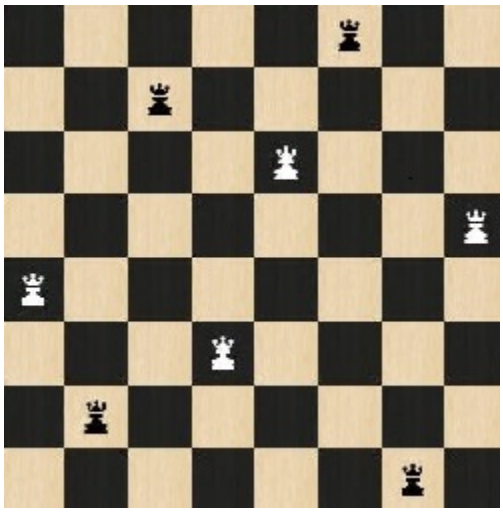
The 8 queens problem

The general state to the "8 queens problem" is to find a way to place eight queens on a 8x8 chessboard so that no queen would attack any other queen. A more mathematical way of expressing the problem would be to place eight queens anywhere on an eight grid where none of them share a common row, column or diagonal.

With such a problem, Genetic Algorithms are adaptative methods which may be used to solve search and optimization problems so they fit perfectly to the 8 queens problem.

Implementation of Genetic Algorithms to the problem

Genetic Algorithms work by applying "Natural Selection" to a population so that only the fittest in the population survive. In our implmentation of the genetic algorithm, members of the population are represented by chromosomes. Each chromosome is a possible board possesion in the 8 queens game. By doing so, the board position is represented as a tuple of 8 numbers, where each represents the position of the queen on each column of the board. For example, the board position below is represented as (4,6,1,5,2,0,7,3):



These "chromosomes" in the population breed among each others and mutate the fittest ones, what creates the solutions with least conflicts among the queens remaining as we could see above. Given a chromosome, the computation of fitness function follows this way:

1st: We compute the number of threaths that the queen in the first column do. The lenght of the chrmosome is N

2nd: We remove the first queen from the array

3td: We compute the number of threaths that the queen in the second column do. The lenght of the chromosome is N-1

4th: We remove the second queen

5th : And so on...

Notice that in order to chose a good starting individual, we avoid to create chromosomes where 2 queens share the same column (i.e: place one queen per cell into the array) and to repeat numbers into the array (i.e: not allowing to share the same row). Given this, the only threads a queen could do are diagonal sided. So it reduces the complexity of the problem.

All in all, the workflow for our Genetic Algorithm is going to folow the folowing path:

1st: Declare necessary variables and initialise the individuals of the current population

2nd: Selection operators through a fitness function

3rd: Crossover operations: to create a new population

4th: Mutation operators: mutate children

So now we are going to explain step by step our algorithm following the defined path above.

Declaring necessary variables and initializing initial population

As the title says, the first part of our GA is to declare all variables we are going to be using. However, the specific values such as the number of queens, probability of mutation and so on will be introduced at the end inside the main function body. That will help when changing parametres in order to get final conclusions or trying different input options.

```
def __init__(self, n, max_population, prob_mutation, number_tournament,tournament_rand,
llargada_one_point_crossover_maxima,crosover_rand,proportion_of_best):

        self.N = n
        self.MAX_POPULATION = max_population
        self.MUTATION_PROBABILITY = prob_mutation
        self.NUMBER_TOURNAMENT_MAX = number_tournament
        self.TOURNAMENT_RAND = tournament_rand
        self.LLARGADA_MAXIMA = llargada_one_point_crossover_maxima
        self.CROSSOVER_RAND = crosover_rand
        self.PROPORTION_OF_BEST = proportion_of_best

def initialize(self):
    self.list_population = []
    i = 0
    for i in range(self.MAX_POPULATION):
        individual = Individual(self.N)
        individual.create()
        self.list_population.append((individual))
        self.AverageFitness()
        i += 1
```

Selection operators

In this step we want to update the fitness of each individual in the population using the fitness function. The general idea behind this step is to select the fittest individuals to simulate the process of "natural selection". There are some different GA strategies and we choosed the "tournment selection". This strategy consists in selecting K individuals from the initial population at random and select the best of these to become parent.

```
def TournamentSelection(self):

        best_fitness = None
        i = 0

        if self.TOURNAMENT_RAND:
            number_tournament = random.randint(1,self.NUMBER_TOURNAMENT_MAX)
        else:
            number_tournament = self.NUMBER_TOURNAMENT_MAX

        for i in range(number_tournament):

            individual = random.choice(self.list_population)

            if (best_fitness == None) or (individual.fitness < best_fitness):

                best_fitness = individual.fitness
                best_individual = individual

            i += 1

        return best_individual
```

Crossover operators

The second step is mixing and matching parts of two parents to form a children. In this specific problem, we select two of the parents and combine them by "breeding". If both parents have common gene's they are also transferred to the child.

```
def CrossOver(self,individual1,individual2):

        if self.CROSSOVER_RAND:
            crosover_point = random.randint(1,self.LLARGADA_MAXIMA)
        else:
            crosover_point = self.LLARGADA_MAXIMA

        genome_individual1 = individual1.genome
        genome_individual2 = individual2.genome

        new_genome_individual1 = [None] * len(genome_individual1)
        new_genome_individual2 = [None] * len(genome_individual1)
```

```
for index in range(len(genome_individual1)):

    if index >= crosover_point:

        new_genome_individual1[index] = genome_individual1[index]
        new_genome_individual2[index] = genome_individual2[index]

new_index_ind_2 = 0
new_index_ind_1 = 0

for gene in genome_individual1:

    if new_index_ind_2 < crosover_point:

        if gene not in new_genome_individual2:

            new_genome_individual2[new_index_ind_2] = gene
            new_index_ind_2 += 1
    else:
        break

for gene in genome_individual2:

    if new_index_ind_1 < crosover_point:

        if gene not in new_genome_individual1:

            new_genome_individual1[new_index_ind_1] = gene
            new_index_ind_1 += 1
    else:
        break

if len(new_genome_individual1) != len(genome_individual1) or
len(new_genome_individual2) !=len(genome_individual1):

    print("Diferencies en la llargada del genoma")
    print(new_genome_individual1)
    print(new_genome_individual2)

    sys.exit()

new_genome_individual1 = self.mutation(new_genome_individual1)
new_genome_individual2 = self.mutation(new_genome_individual2)

new_genome_individual1 = tuple(new_genome_individual1)
new_genome_individual2 = tuple(new_genome_individual2)


child_1, child_2  = Individual(self.N), Individual(self.N)

child_1.assign(new_genome_individual1)
child_2.assign(new_genome_individual2)

return (child_1,child_2)
```

The crossover it has been choosen is the so-called one point crosover. That is, the algorithm splits parent's chromosome in two parts. Then from the father it places the first half into the first positions of the child number one, and the second half in the latest positions of the child 2, maintaining the position of the chromosomes from the father to the child. Then it takes from the mother the rest of the genes in order that child 1 do not take from the father. And it do the same for the second son, but taking the first genes from the mother. With this it avoids to repeat genes from one generation to the other.

Mutation operators

Mutation can be explained as a small random tweak in the genes of the individual in order to achive a variation of the original individual. In other words, it is used to introduce diversity in the genetic population and is a key concept related to the "exploration" of the search space. In our problem, the genetic mutation is introduced to the chromosome according to the prefered mutation probability, normally a low probability and in our case we used a 3% probability of mutation. In our code, it is restricted in 1 the number of a chromosome can mutate.

```
def mutation(self, genome_individual):

        new_genome_individual = list(genome_individual)

        rand_value = random.uniform(0,1)

        if rand_value < self.MUTATION_PROBABILITY:

            rnd_index_1 = random.randint(0,self.N - 1)
            rnd_index_2 = random.randint(0,self.N - 1)

            new_genome_individual[rnd_index_1] = genome_individual[rnd_index_2]
            new_genome_individual[rnd_index_2] = genome_individual[rnd_index_1]

        return new_genome_individual
```

Relation between Mutation and Crossover.

In genetic algorithm there are two principal operators to the population; Crossover operators and Mutation operators. These two are the central part of the algorithm and are directly related to the exploratory part of the algorithm (Mutation operator) and with the exploitatory (Crossover operation). If these two are not well balanced, then we get two main ends for the algorithm.

$$\uparrow\uparrow Mutation \rightarrow No\ convergence$$

$$\uparrow\uparrow Crossover \rightarrow Convergence\ to\ local\ minimum$$

This schema says that if in our algorithm we have the mutation rate high enough, we won't get any solution, or if we find one, it will be totally randomly (like montecarlo algorithm would do). On the other hand, if we increase crossover rate enought, we will get stuck in a local minimum for the fitness function.

However, tunning the parameters properly we can obtain an algorithm which has a significant rate of convergence but avoids getting stuck into a local minimum because of the mutation rate.

Results of the program

Ensambling all the previous pieces and executing the main body function below we get one of the solution for the 8 queens problem, where:

"n" is the dimension of the grid. "max_population" is the maximum population. "prob_mutation" is the mutation probability. "number_tournament" is the number of tournments to choose parents. "tournament_rand" For each torunment: if TRUE, the number of each comparision lies between 1 and "number_tournament", if FLASE the number of comparisions is "number_tournament". "llargada_one_point_crossover_maxima" is the genome position where crossover takes place. "crosover_rand". If TRUE, in each crossover the breakpoint it's a random number between 0 and "llargada_one_point_crossover_maxima".If FALSE, it's going to be always "llargada_one_point_crossover_maxima". "proportion_of_best" is the proportion of the sorted population we want to use to calculate theaverage fitness function. Just to see visualize when executing how the fitness function evolves in each iteration.

```
##############################################
        # Body of the program #
##############################################

poblacio_total = Population(
    n = 8
    max_population = 100,
    prob_mutation = 0.03,
    number_tournament = 99,
    tournament_rand = True,
    llargada_one_point_crossover_maxima = 7,
    crosover_rand = True,
    proportion_of_best = 0.2)


poblacio_total.initialize()
i = 0

for i in range(40000):

    poblacio_total.NextGeneration()
    individual_solution = poblacio_total.BestIndidivualSoFar()
    if i%1 == 0:
        print(poblacio_total.fitnessavg)

    if individual_solution != None:

        print(individual_solution.genome)
        print("Solution found!!!")
        break

    i += 1
```

In that case, one of the posible solutions was (3,6,4,1,5,0,2). As in a computer science lab, from this point we started to run the algorithm several times to get some of the 92 possible solutions and prove the algorithm

works.

X0	X.2..7..3..6..0..5..1..4.
1	(3, 6, 2, 7, 1, 4, 0, 5)
2	(3, 0, 4, 7, 5, 2, 6, 1)
3	(4, 6, 3, 0, 2, 7, 5, 1)
4	(1, 6, 4, 7, 0, 3, 5, 2)
5	(3, 7, 0, 2, 5, 1, 6, 4)
6	(3, 1, 7, 4, 6, 0, 2, 5)
7	(2, 7, 3, 6, 0, 5, 1, 4)
8	(3, 5, 7, 2, 0, 6, 4, 1)
9	(5, 1, 6, 0, 3, 7, 4, 2)
10	(2, 4, 1, 7, 5, 3, 6, 0)
11	(3, 5, 7, 2, 0, 6, 4, 1)
12	(4, 1, 5, 0, 6, 3, 7, 2)
13	(1, 5, 7, 2, 0, 3, 6, 4)
14	(4, 0, 7, 3, 1, 6, 2, 5)
15	(5, 2, 4, 7, 0, 3, 1, 6)
16	(5, 7, 1, 3, 0, 6, 4, 2)
17	(2, 5, 7, 1, 3, 0, 6, 4)
18	(3, 5, 7, 2, 0, 6, 4, 1)
19	(2, 0, 6, 4, 7, 1, 3, 5)
20	(4, 7, 3, 0, 2, 5, 1, 6)
21	(3, 5, 7, 2, 0, 6, 4, 1)
22	(5, 2, 6, 1, 7, 4, 0, 3)
23	(2, 5, 1, 4, 7, 0, 6, 3)
24	(5, 2, 0, 6, 4, 7, 1, 3)
25	(4, 6, 1, 3, 7, 0, 2, 5)
26	(7, 1, 4, 2, 0, 6, 3, 5)
27	(6, 0, 2, 7, 5, 3, 1, 4)
28	(4, 1, 3, 6, 2, 7, 5, 0)
29	(3, 1, 7, 5, 0, 2, 4, 6)
30	(1, 4, 6, 0, 2, 7, 5, 3)
31	(3, 5, 7, 1, 6, 0, 2, 4)
32	(6, 1, 3, 0, 7, 4, 2, 5)
33	(2, 5, 3, 0, 7, 4, 6, 1)
34	(4, 6, 1, 3, 7, 0, 2, 5)
35	(5, 2, 6, 1, 7, 4, 0, 3)
36	(1, 6, 2, 5, 7, 4, 0, 3)
37	(5, 3, 0, 4, 7, 1, 6, 2)
38	(2, 5, 1, 6, 0, 3, 7, 4)
39	(6, 2, 0, 5, 7, 4, 1, 3)
40	(3, 6, 0, 7, 4, 1, 5, 2)
41	(5, 2, 6, 3, 0, 7, 1, 4)
42	(5, 3, 6, 0, 7, 1, 4, 2)
43	(4, 0, 7, 5, 2, 6, 1, 3)
44	(3, 6, 0, 7, 4, 1, 5, 2)
45	(6, 3, 1, 4, 7, 0, 2, 5)
46	(4, 7, 3, 0, 2, 5, 1, 6)
47	(0, 4, 7, 5, 2, 6, 1, 3)
48	(5, 2, 0, 6, 4, 7, 1, 3)
49	(2, 5, 7, 1, 3, 0, 6, 4)

In the following part of the report we will anlayze the performance changing parameters and incresing the number of queens to push the algorithm to the limit.

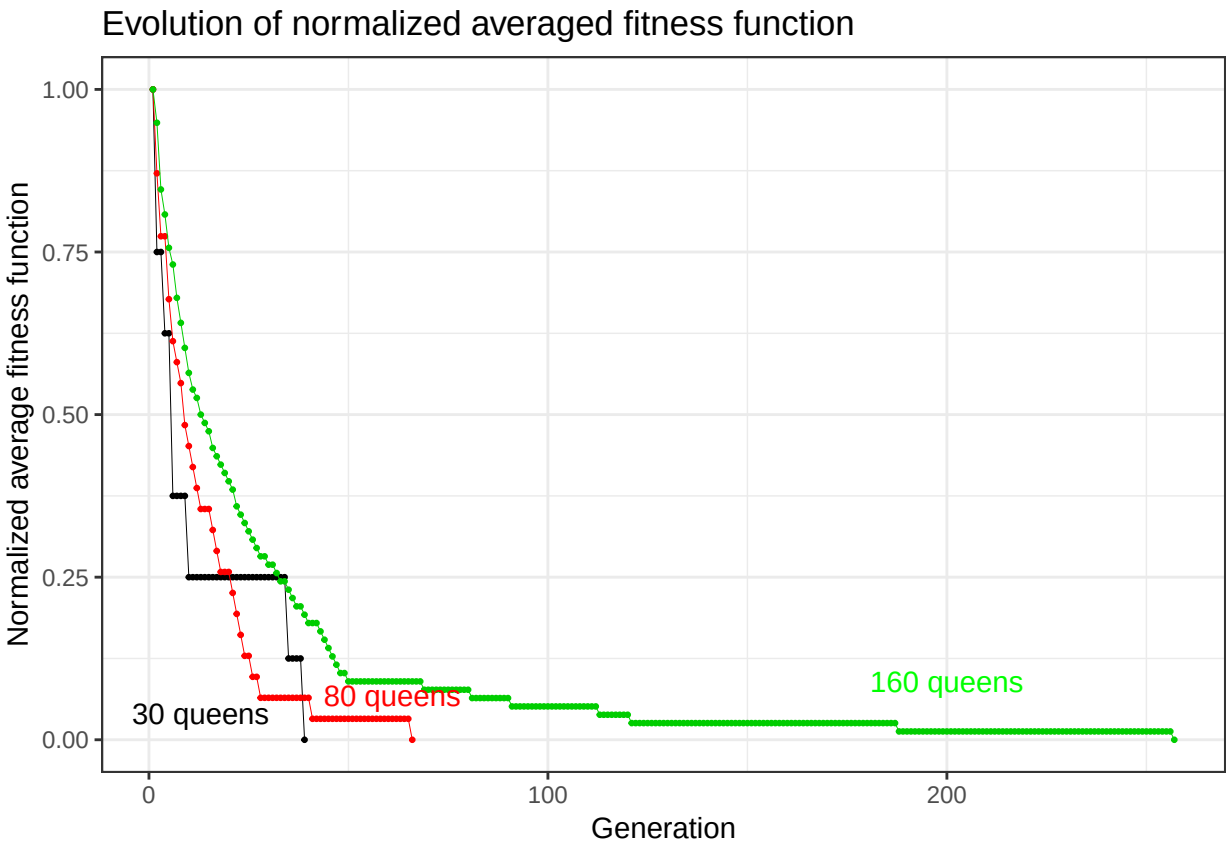
Performance analysis

At this point we have developed a genetic algorithm able to solve the 8 queens problem. However, we saw that solving the 8 queens problem is not a real deal for it, since it gets the solution in the first generations. So in order to understand a little bit more the algorithm and get a real perspective of it's power we have used bigger number of queens to appreciate the effects on the average fitness function through each generation.

we increased the number of queens for those specific parameters:

queens	max_population	mutation	number_tournments	tournment_rand	generations_needed
8 queens	1000	0.13	800	False	1
30 queens	1000	0.13	800	False	39
80 queens	1000	0.13	800	False	66
160 queens	1000	0.13	800	False	257

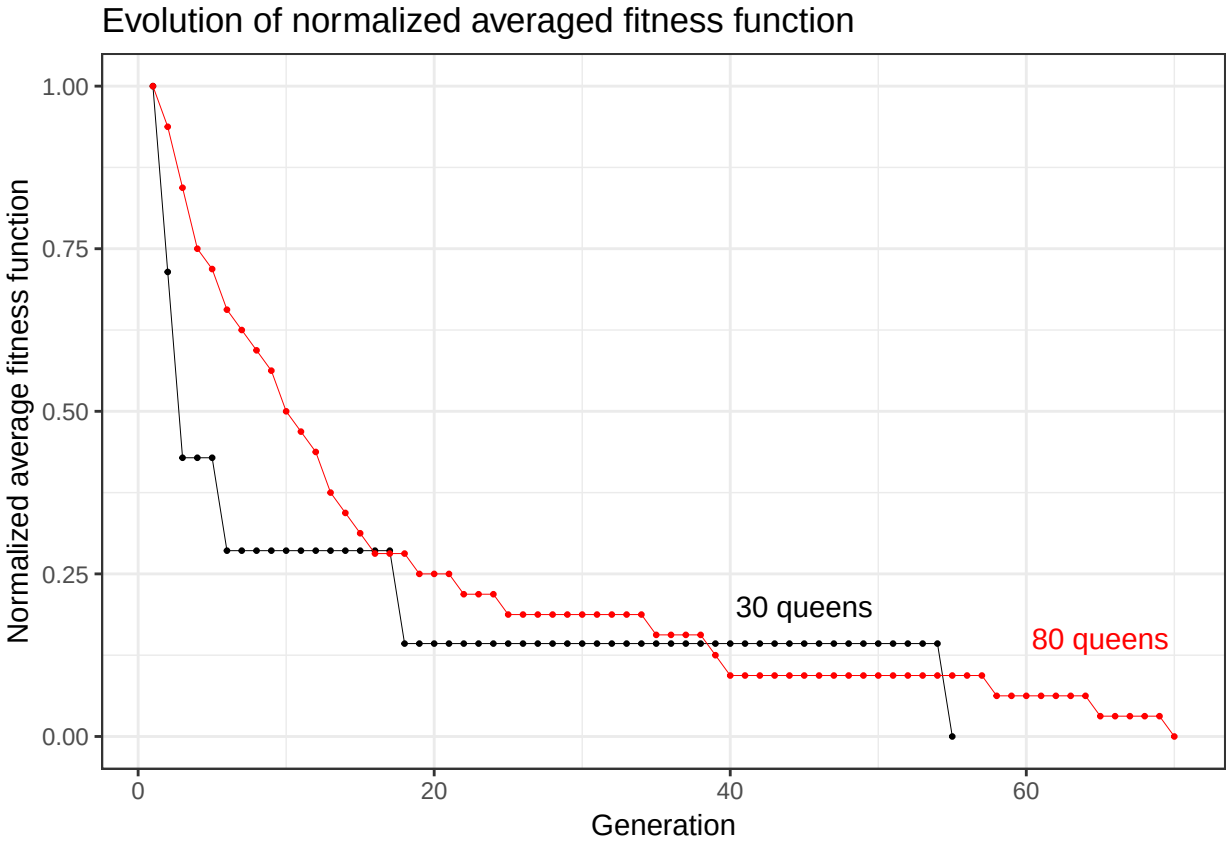
The time and generations needed to converge increases as we increase the number of queens, which it's totally expected. Then for those results we also obtained the average of the fitness function in each generation in order to visualize the smoothness of the curve when getting close to the solution. Since the solution for 8 queens is obtained at first generation we plot the averages of the fitness functions when performing the algorithm for higher queens:



And for the same number of queens, changing parameteres like the mutation probability we obtain:

queens2	max_population2	mutation2	number_tournments2	tournment_rand2	generations_needed2
8 queens	1000	0.03	800	False	1
30 queens	1000	0.03	800	False	56
80 queens	1000	0.03	800	False	70

As in the previous executions, time and generations increase with the number of queens as we should expect. However, when changing the mutation rate to 3% the algorithm needs more generations to find the solution.



Conclusions

In this work we have developed a genetic algorithm able to solve the n queens problem and able to find the solutions for the specific 8 queen problem. Morover we found solutions for multiple n queens problems and dealing with the differences when changing parameters as the mutation rate. As we can see, GA is so useful and powerful for optimization, that is, so find a one solution for the specific problem, no matter what solution is.

However, the importance of knowing the problem remains relevant because it is related with the parameters of the problem, for instance, the amount of population and the representation of the genome. One of the most important things we learned in our GA the fintess function depends on the position of the queen. The position can be represented in two ways, with an array of rows and columns or with an array of "n" longitude, where the position of the array is represented by the value on the column and the value of the array represents the row. The way we represent the crossover is directly related to the form we choose to represent the position in one of the options explained before. If we don't take into account in the crossover the actual representation of the genome then you get a functional algorithm, with correct functions for mutation and crossovers but will never converge because is never creating well fitted individuals each for each generation.