RESEARCH AND INNOVATION

GENETIC ALGORITHMS – 8-QUEENS PROBLEM

Authors: Adria Fenoy & Lawrence Adu-Gyamfi
Date: 30/11/2018

Table of Contents

1. INTRODUCTION

1.1. Genetic Algorithms Overview

Genetic algorithms are part of a general bigger family of evolutionary algorithms that are useful for solving search and optimization problems that have large search spaces. It is able to arrive at a solution by continuously generating candidate solutions, evaluating how well the solutions fit the desired outcomes. The algorithm continues to improve the best solutions at every until it arrives at the best solutions. This methodology is similar to the phenomenon of evolution based on the principles of natural selection.

A major underlining feature of genetic algorithms is the fitness which serves as a feedback mechanism to the genetic engine and helps in making the better choice between two potential solutions. It combines the benefits of exploratory and exploitatory algorithms as part of its operation.

The typical working principle of genetic algorithms is outlined below:

1. Generate initial population of individuals based on the population size, with each representing a possible solution. This can be done randomly
2. Assess the fitness of each individual in the population and assign each individual a fitness score based on how well they solve the problem.
3. From the parent population, generate a new population of children solutions of **same population size** as parents as follows:
    a. Select 2 parents from the parent population, with a bias towards fitter individuals, for reproduction. This is done with replacement.
    b. **Crossover** the two parents to produce two new children with a probability known as crossover rate. Normally this should more or less result in fitter children or better solutions.
    c. Perform a **mutation** operation on the two new children (solutions) based on a probability known as the mutation rate.
    d. Add these new children to the new population.
4. Replace the parent population with the new population. (which should normally be a population of fitter (better) solutions.
5. Repeat steps 2 -4 until the ideal solution is realized or the population converges (about 95% of the solutions in the population have the same makeup), or the maximum number of generations, if there is one, is reached.

A pseudocode of the algorithm is shown below: Ref. [3]:

```
popsize ← desired population size

P ← {}
for popsize times do
    P ← P ∪ {new random individual}
Best ← □
repeat
    for each individual P_i ∈ P do
        AssessFitness(P_i)
        if Best = □ or Fitness(P_i) > Fitness(Best) then
            Best ← P_i
    Q ← {}
    for popsize/2 times do
        Parent P_a ← SelectWithReplacement(P)
        Parent P_b ← SelectWithReplacement(P)
        Children C_a, C_b ← Crossover(Copy(P_a), Copy(P_b))
        Q ← Q ∪ {Mutate(C_a), Mutate(C_b)}
    P ← Q
until Best is the ideal solution or we have run out of time
return Best
```

1.2. Definition of Problem (8 – Queens Problem)

The report focuses on the solution of the 8-queens problem using a classical genetic algorithm.
The 8-Queen problem is one specific form of the general N-Queen problem which can be defined as finding the positions of 8 queens on an 8 x 8 chessboard while ensuring no two of the queens are attaching each other.

According to Wikipedia Ref. [1], this problem has a maximum of 92 solutions and this is reduced further to 12 fundamental solutions if symmetry operations such as rotation and reflection are counted as one solution.
Some sample solutions are shown below.

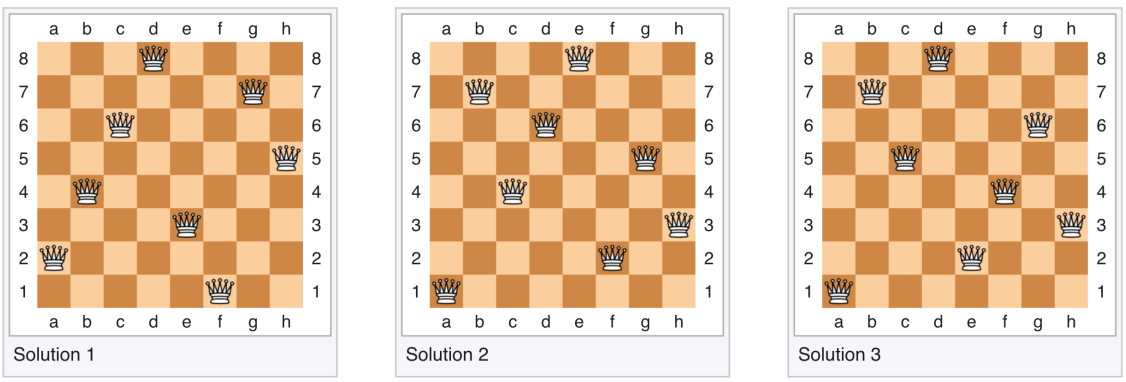


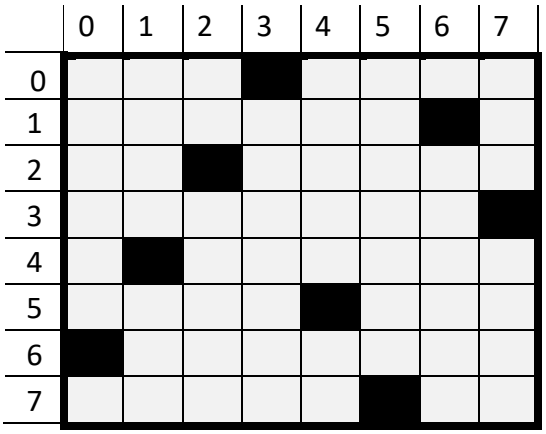
Figure 1-1: Sample Solutions of the 8-Queen Problem. Ref. [1]

2. METHODOLOGY

This section explains the methodology that has been implemented using the genetic algorithm in finding solutions to the 8-queen problem.
The solution has been implemented using the python programming language.

2.1. Individual

An individual or a candidate solution for this problem is represented as a list of N positions on an N x N chessboard. N in this case is 8 for the 8-queen problem.
The index of each position automatically represents its row on the chessboard.



Positions : [3 , 6 , 2 , 7 , 1 , 4 , 0 , 5]

Figure 2-1:Problem-specific Definition of Individual Solution

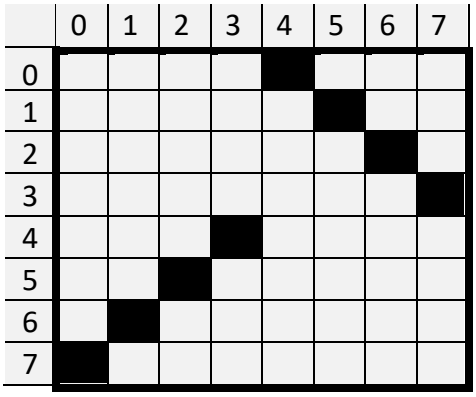
2.2. Population

A population in the solution to the 8 Queens problem is represented by a set of individuals or solutions- with each solution representing a list of positions of the 8 Queens on the chessboard.
The initial population of the parents are generated randomly based on the population size specified and taking into account the restrictions of the genotype as described in section 2.1.

2.3. Fitness function

The fitness of each individual solution is calculated with the number of attacks between queens, meaning that the **optimal fitness is 0.**
This is implemented in the code by checking at each queen in position “i”, and check the queens that are in the same diagonal for positions “i+1”, “i+2”, … , “n”. This check is performed for both diagonals on the board.

Due to the way the individuals are generated, it is always guaranteed that each queen position will have a unique row, and column. Consequently, this is not checked in the determination of the fitness.



Positions : [4 , 5 , 6 , 7 , 3 , 2 , 1 , 0]

Figure 2-2: Typical Layout of Individual Solution

Fitness counter is incremented based on the number of similar contiguous numbers.

Diagonal_1 = Positions + Index

[4,6,8,10,7,7,7,7] = [4,5,6,7,3,2,1,0] + [0,1,2,3,4,5,6,7]

3+2+1+0 = 6

Diagonal_2 = Positions – Index

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

3+2+1+0 = 6

Total Fitness = 12

2.4. Selection Methods

The parents of each population are selected for recombination (crossover and mutation) randomly based on a selection principle with more likelihood to favour the fittest individuals.

The tournament selection algorithm has been tested in selecting parents for crossover as described below.

2.4.1. Tournament Selection

The basic form of the tournament selection algorithm is used in selecting parent solutions for recombination. Pairs of parents are picked from the population, one at a time at random (with replacement), based on the best fitness out of a specified tournament size.

The pseudocode of the algorithm is shown below: *Ref. [4].*

```
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
```

It is implemented in our program with the following code:

```
fitness = fitness(individuals)
positions = [ sorted(np.random.choice([0:n], tournament_size))[0]
              for _ in range(2*self.n) ]
return [ (individuals[i], individuals[j])
          for i, j in zip(positions[0::2], positions[1::2]) ]
```

First of all, we compute the fitness, and sort the individuals by fitness (this is done inside the fitness function). Then we select a number of positions given by the tournament size and select the best position. Because of the individuals are sorted, this position will correspond to the individual in those positions with the best fitness. We do this procedure 2n times, where n is the size of the whole population. The we group half of the positions in this list with the other half and select the individuals corresponding to it. These individuals will be the parents for crossover algorithm.

2.5. Crossover Algorithm

To perform crossover, not all individuals are allowed, so we have certain restrictions. Each individual must have no repeated positions in each column to avoid that there are queens in the same row or column.

To achieve crossover satisfying this condition, we use cycle crossover implemented the following way:

```
son1, son2 = deepcopy(ind1), deepcopy(ind2)
start = np.random.choice([0:dim], 1)
i = start
while True:
    son1.values[i], son2.values[i] = son2.values[i], son1.values[i]
    i = np.where(ind1.values == ind2.values[i])
    if i == start: break
return son1, son2
```

First of all, we create sons that are identical to their parents. Then we select one random position to start looping, and we swap values of both sons in this position. After performing the swap, we look at the parents to know which value has been swapped from the second parent and look for this value in the first parent; this will be the next swapped position. The loop finishes when the cycle is closed, i.e. when the current position is the starting one.

2.6. Mutation

2.6.1. Mutation Algorithm

To implement mutation in our genetic algorithm, we select some specific genes on each individual and perform a derangement with them. The selection of those genes is explained in section 2.6.2.

So, assuming that we have a given set of positions p = {p1, p2, ..., pn} we want to perform the derangement, we use the following algorithm:

```
p = [p1, p2, ... , pn]
swapped = []
for i in positions[:-1]:
    swapped += i
    j = np.random.choice([k for k in positions if k not in swapped], 1)
    values[j], values[i] = values[i], values[j]
```

Swapped is initialized as an empty list which will contain one of the positions that will be swapped each time. Then, we start iterating over all positions except the last one (because the second last one will already perform a swap with the last one).

We start adding the current position to the swapped list and, using ***np.random.choice,*** we select another element from positions which has not been swapped. After this, we swap both elements and repeat the algorithm for next position.

2.6.2. Algorithm for Selection for Mutation

To determine the positions that will be swapped for each individual, we implement a function that chooses them randomly, according to some probability of performing mutation. We also add the possibility that more than two positions are selected for

mutation, but it is very small, it's twice less probable to perform mutation on three positions than performing mutation in two positions. All of this is implemented with the following algorithm:

```
probability = [1-p] + [ p*(1/2)**(i+1) for i in [0:dim-2] ] + [(1/2)**(dim-2)]
num_mutations = [ np.random.choice([1:(dim+1)], 1, p = probability ) for i in [0:n] ]
return [ np.random.choice([0:dim], n) for n in num_mutations ]
```

Probability is a list with the probabilities of swapping 1 element (which is the same as not swapping), 2 elements, 3 elements, and so on. Then, we determine the number of mutations that each individual will suffer according to the probabilities and, finally, we choose the positions that will be swapped according to the number of mutations for each individual.

2.7. Convergence

The convergence of the population is monitored in comparison with that of the fittest individual, as the generations progress. It is expected that as the population converges the average fitness will approach that of the best individual. Ref. [2]
We have checked for convergence in our code by ensuring a minimum of 95% of the population have a similar fitness as that of the fittest individual (fitness of 0) before breaking out of the outer loop.

3. RESULTS

This section presents some results of running our implementation of the genetic algorithm towards solving the 8-queen problem. The main analysis is done with results for the 8-queen. Additionally, some results of running the program with higher dimensions is shown to verify the effectiveness of the implementation.

3.1. Fitness Evolution over Generations

The following graphs demonstrate evolution of the population towards the optimal fitness with respect to that of the best individual.
We have experimented with several values for the rates of crossover, mutation and the size for the tournament selection for crossover.
With a population size of 20 individuals, and crossover and mutation rates of 0.8 and 0.01 respectively, we noticed that it took much longer for before the program found a solution and converged (>95% or population with optimal fitness). Analyzing the graph, we conclude that the solution gets stuck in a local minimum for some time before at some point a mutation occurs leading to the solution and eventual convergence. We can say the algorithm is being more exploitatory in this case. The results are shown below.

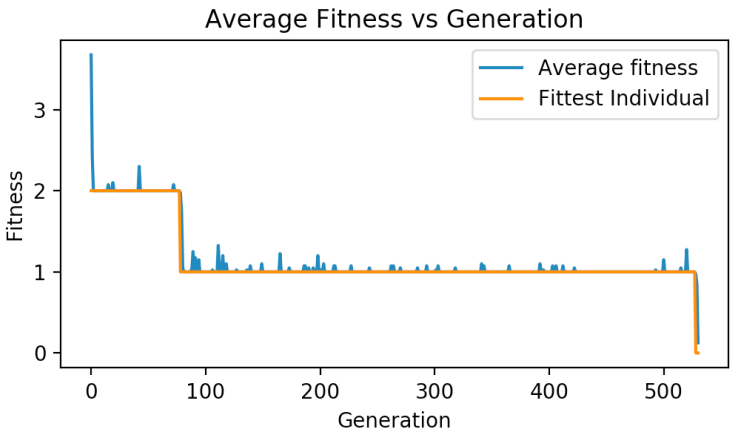
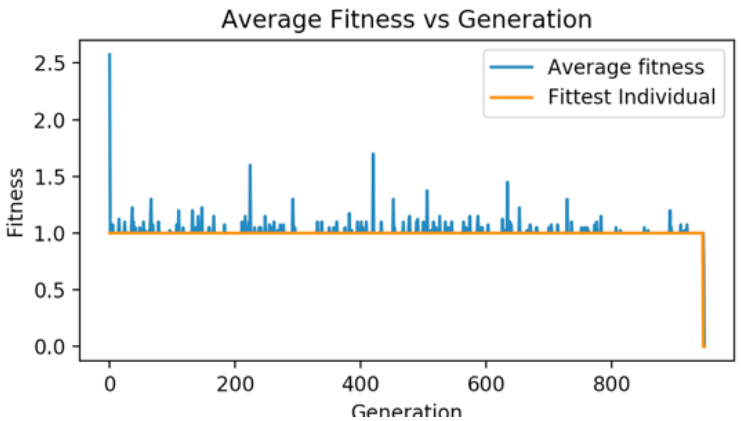
Population size = 20 Mutation rate = 0.01 Crossover rate = 0.8 Tournament size = 5	
Population size = 20 Mutation rate = 0.01 Crossover rate = 0.8 Tournament size = 7	

Table 3-1: Fitness Evolution (Base Case)

Consequently, we increased the mutation rate to allow the algorithm to be more exploitatory.
This we noticed caused the algorithm to converge over an evolution of fewer generations. A further increase in the mutation rate caused it to converge even more quickly.

The results are shown below:

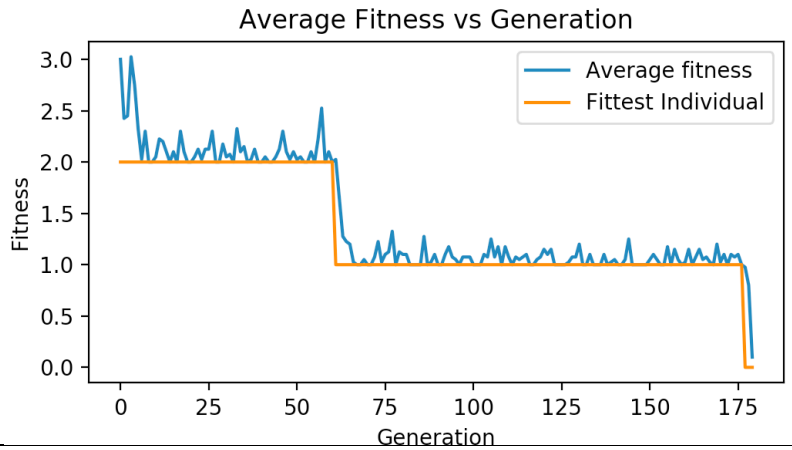
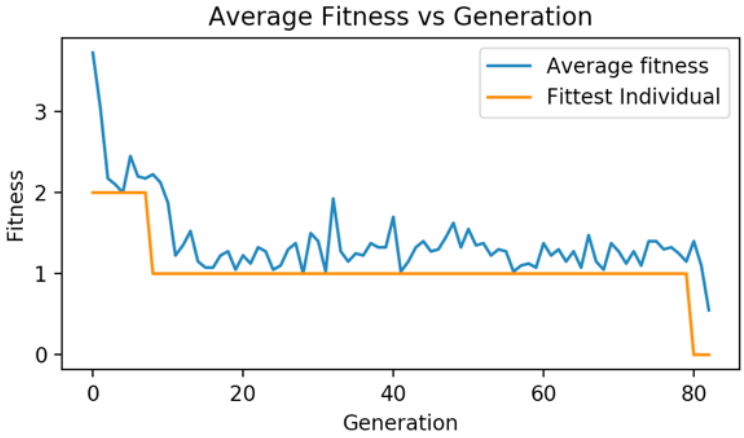
Population size = 20 Mutation rate = 0.1 Crossover rate = 0.8 Tournament size = 5	
Population size = 20 Mutation rate = 0.3 Crossover rate = 0.8 Tournament size = 5	

Table 3-2: Fitness Evolution (Increased Mutation Rate)

The following results show the effect of increasing the tournament size for the selection for crossover.

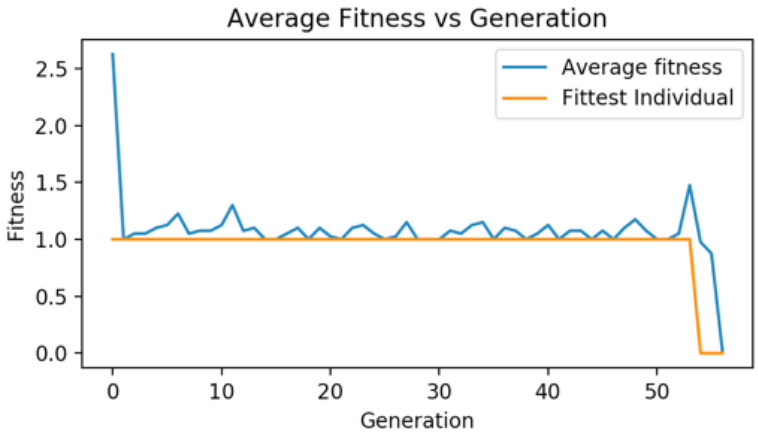
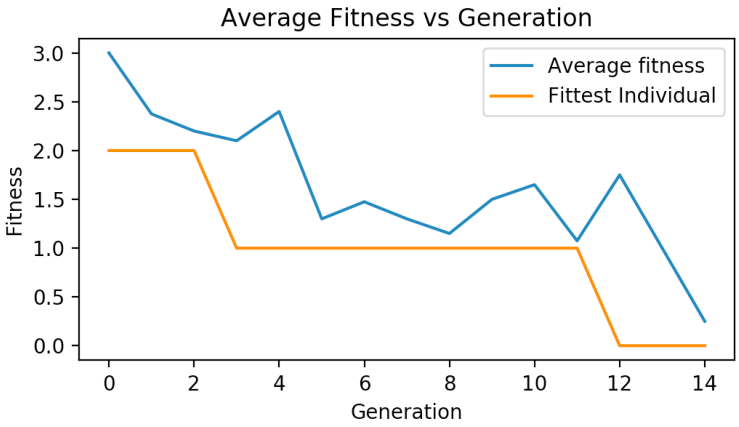
Population size = 20 Mutation rate = 0.1 Crossover rate = 0.8 Tournament size = 7	
Population size = 20 Mutation rate = 0.3 Crossover rate = 0.8 Tournament size = 7	

Table 3-3: Fitness Evolution (Increased Tournament Size)

It is noticed that, it takes fewer generations for the algorithm to converge towards the optimal fitness.

This is in line with the working principle of the tournament selection algorithm. Increasing the tournament size increases the selection pressure meaning there is more likelihood to have individuals of higher fitness during the random selection meaning less probability to have individuals of lesser fitness eventually in the mating pool. Ref. [3]

3.2. Sample Solutions for 8-Queens

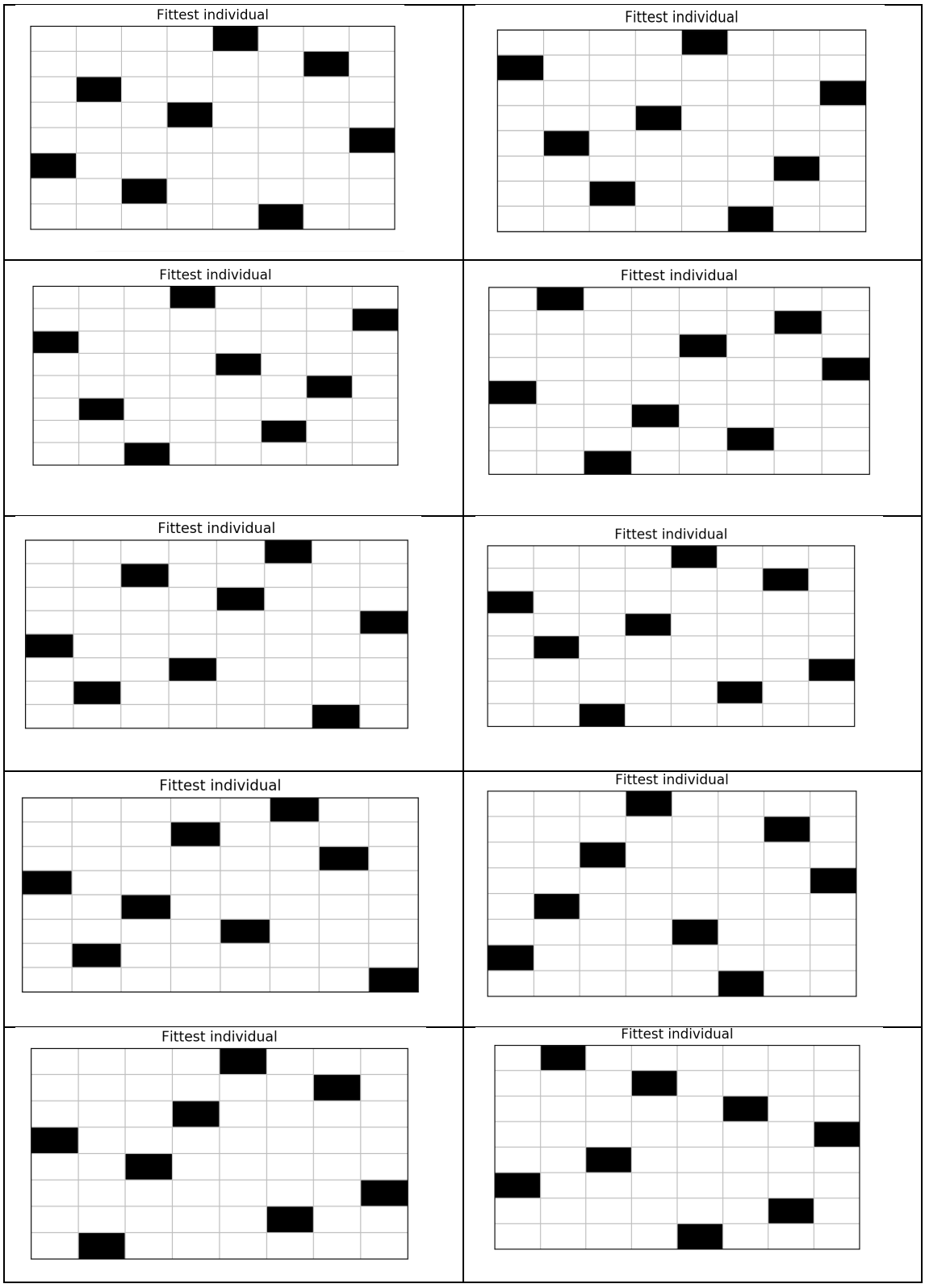


Table 3-4: Sample Solutions for 8-Queen Problem

3.3. Solution of Higher Dimensions

To further verify the effectiveness of our implementation of the genetic algorithm, we have tried the program on bigger dimensions of the chessboard. Following are some results of this experimentation.

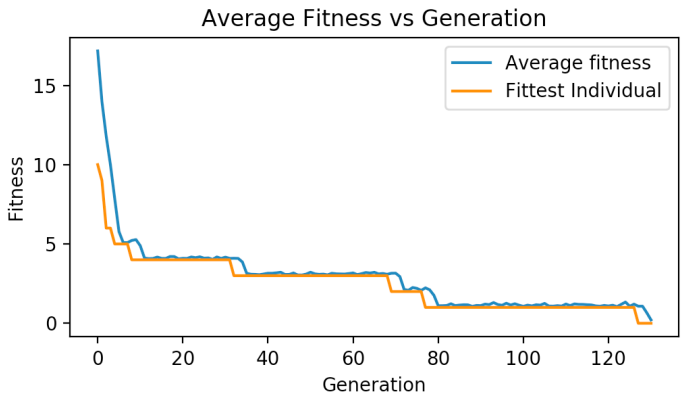
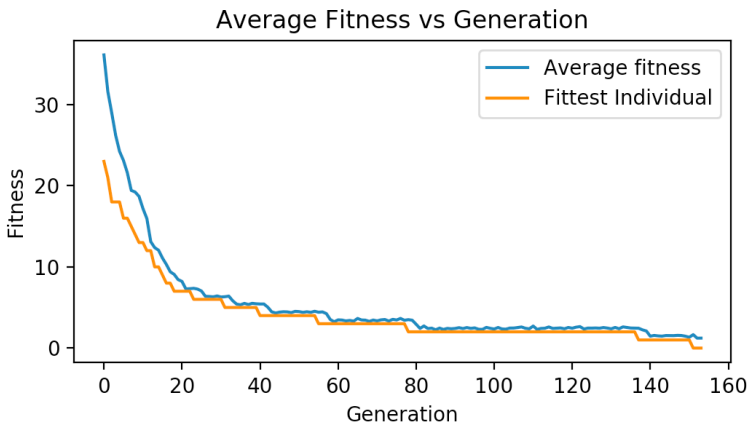
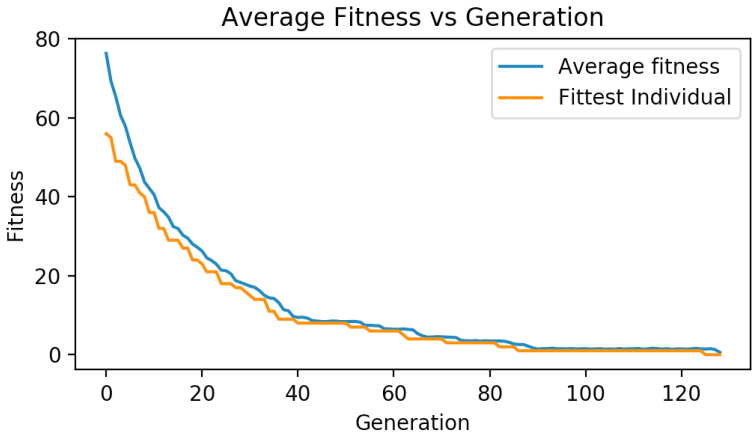
Population size = 200 **No. of Queens = 32** Mutation rate = 0.1 Crossover rate = 0.8 Tournament size = 7	
Population size = 200 **No. of Queens = 64** Mutation rate = 0.3 Crossover rate = 0.8 Tournament size = 7	
Population size = 400 **No. of Queens = 128** Mutation rate = 0.3 Crossover rate = 0.8 Tournament size = 7	

Table 3-5: Fitness Evolution (Problem Sizes 32, 64, 128)

3.3.1. Sample Solution – 32 x 32

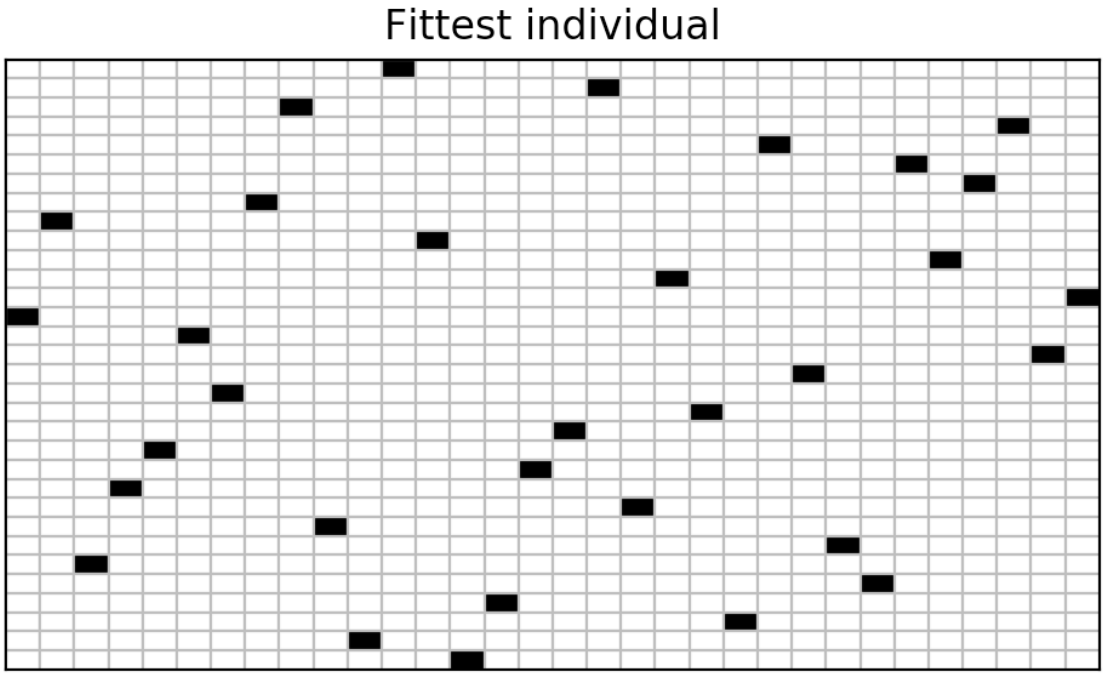


Figure 3-1: Sample Solution – 32 x 32

3.3.1. Sample Solution – 64 x 64

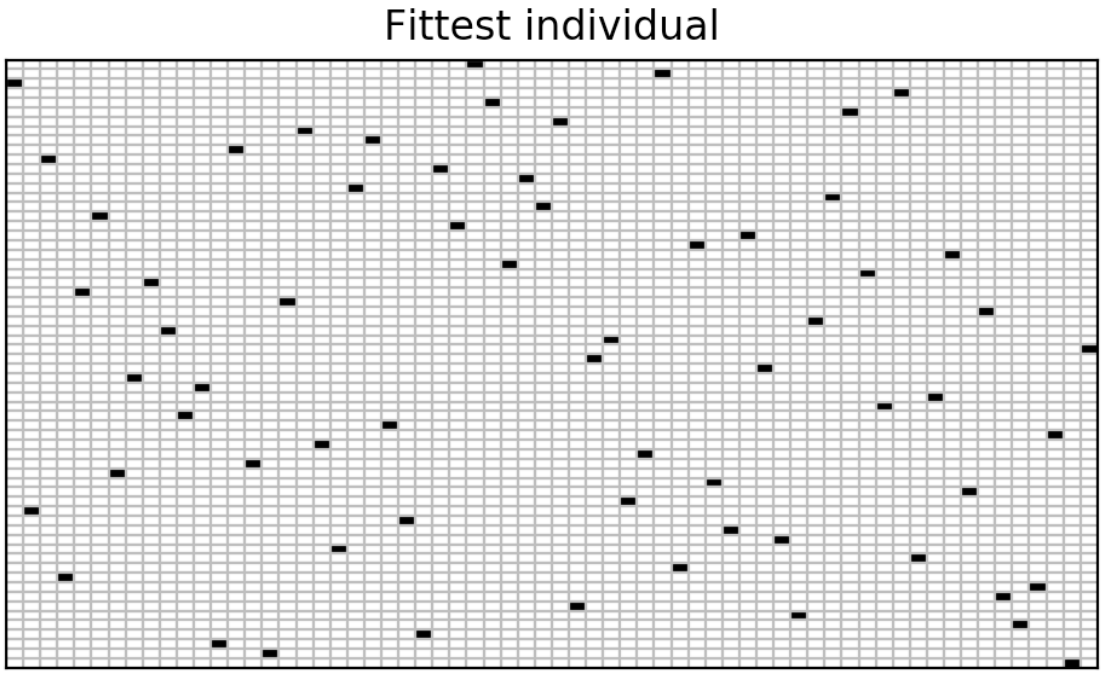


Figure 3-2: Sample Solution – 64 x 64

4. CONCLUSION

In conclusion, through this project as captured in this report, we have managed to implement the genetic algorithm taking to solve the 8-queens problems and even for higher dimensions of problem size. We have utilized the several principles of genetic algorithm including population definition and selection, recombination through crossover and mutation and studied as well the convergence of the population of solution over generations.

The definition of an individual solution in this implementation was important as it greatly assisted in the definition of the fitness function. By choosing to define the genotype of a solution as only the number of queens, we simplified the fitness function and avoided having any individuals in similar rows or columns.

In determining the best parameters to choose for the rates of crossover and mutation, as well as the size of the tournament for the selection for crossover, we confirmed that there is no hard and fast rule concerning the final choice. Several tries had to be made, together with a careful analysis of the resulting fitness evolutions of the generations to identify which parameters to tune in order to make the algorithm efficient.

As shown in the graphs, particularly for the problem size of 8 queens, we notice the issue of premature convergence sometimes where most of the individuals in the generation converge around the solution with the highest fitness, thereby limiting the possibility to find new and better solutions. We notice that the search for the optimal fitness continues over several generations before eventually a mutation leads to arriving at a better solution and eventual convergence. All these have been taken into account in arriving at the parameters used finally to get the better results.

5. REFERENCES

The following references have been used in one way or the other in preparation of this report. They have been referenced as and when applicable.

1. https://en.wikipedia.org/wiki/Eight_queens_puzzle
2. An Overview of Genetic Algorithms, *David Beasley, David R. Bull, Ralph R. Martin*
3. Introduction to Genetic Algorithms and their Applications (A lectures for the Autonomous University of Barcelona), *Jacek Dziedzic*
4. Essentials of Metaheuristics, A Set of Undergraduate Lecture Notes, *Sean Luke*
5. An Introduction to Genetic Algorithms, *Melanie Mitchel*