Solving N-Queens Problem Using Genetic Algorithm

FISAC - Soft Computing Paradigms

Member Details:
Saheba Patney - 210911060
Harshita Gupta - 210911224
Mahi Simhachal Reddy - 210905252
Sidhartha Perla - 210953084
Pranav Yarasi - 210911376

Contents

1 Introduction

The n-queens problem, initially introduced in 1850 by Carl Gauss, may be stated as follows: find a placement of n queens on an n×n chessboard such that no queen can be taken by any other. While it has been well known that the solution to the n-queens problem is n. Many of these solutions rely on a specific formula for placing queens or transposing smaller solution sets to provide solutions for larger values of n (Bernhardsson, 1991 Hoffman et al., 1969). Empirical observations of smaller-size problems show that the number of solutions increases exponentially with increasing n. Alternatively, search-based algorithms have been developed. Several researchers and scientists offered various Heuristic algorithms for optimisation by modelling from physical and biological processes in nature, which often operate collectively to overcome this problem. We have used the genetic algorithm to solve this problem. According to Darwin's theory, there are three stages of the genetic model: (1) natural selection, (2) crossing, and (3) mutation. The first two stages are evident for every new breed, but the last is done if only the breed fails to survive or violates N-Queens rules. Then, Mutation is done to make it fit to survive.

2 Problem Statement

Solving the N-Queen problem using Genetic Algorithm

The aim of the N-Queens Problem is to place N queens on an N x N chessboard in a way so that no queen conflicts with the others.

3 Methodology

To solve the N-Queens problem, we use several Genetic Algorithm Components:

1. Chromosome Representation: Each chromosome represents a potential solution: a queen's placement on the chessboard. We can represent this as an array or a permutation, where each element represents the column index of the queen in a particular row. For example, [2, 6, 3,6,1,2] represents a placement where the queen in the first row is in column 2, the queen in the second row is in column 6, and so on. This is shown in Figure 1.

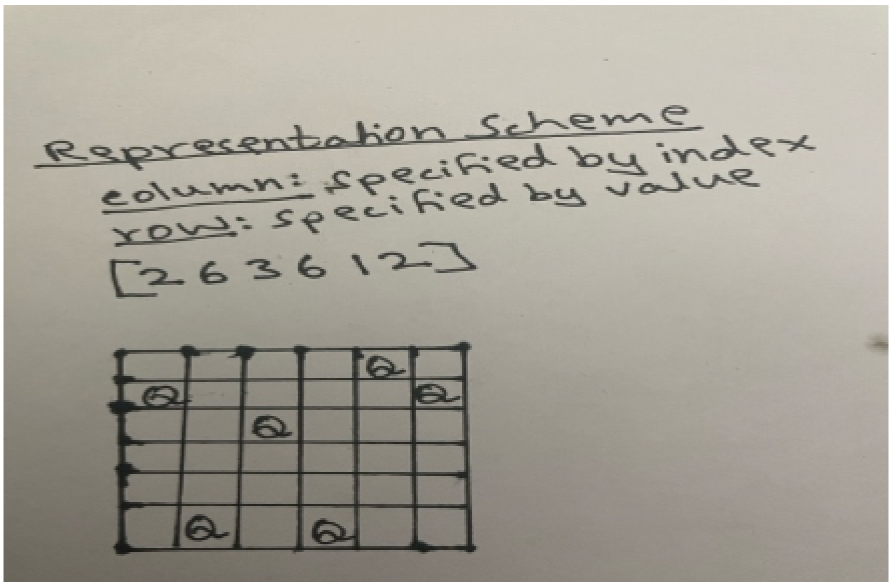


Figure 1: Representation Scheme Column

2. Initialisation: Randomly initialise a population of chromosomes, where each chromosome represents a randomly valid placement of queens on the chessboard. Ensure that no two queens share the same row or diagonal.

3. Selection: Select chromosomes from the population based on their fitness. In this case, fitness can be defined as the number of conflicts (pairs of queens threatening each other) in the placement. Higher fitness means fewer conflicts. Use a selection mechanism such as roulette wheel selection to select parents for reproduction, with probabilities proportional to their fitness.

4. Crossover: Perform crossover (recombination) to create offspring by combining genes (queen placements) from selected parent chromosomes. Crossover methods can be used for permutation-based representations, such as order crossover or cycle crossover. Figure 2 shows the crossover step.

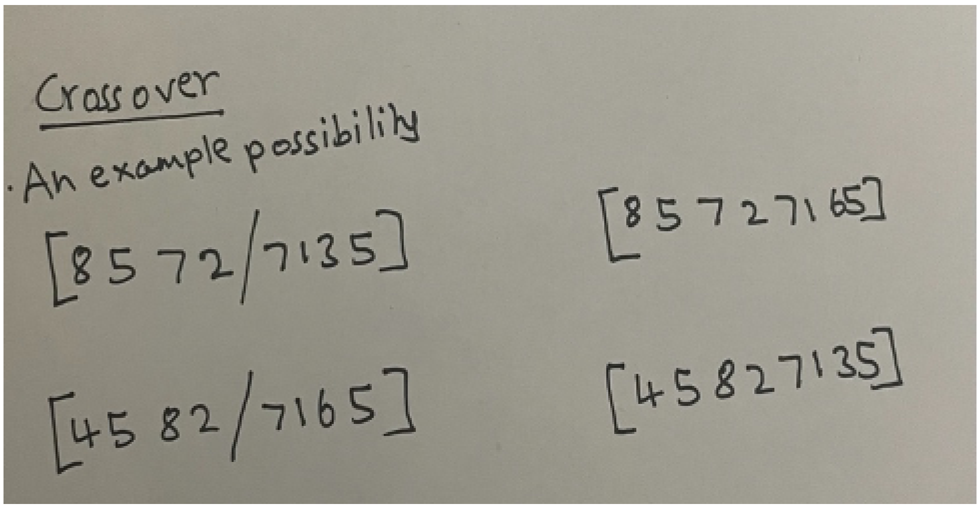


Figure 2: Crossover

5. Mutation: Introduce random mutations to maintain genetic diversity. Mutations in this context involve swapping the positions of queens in the placement to explore new solutions. For example, randomly select two positions in the placement and swap the queens at those positions. Another method would be to flip the bits of numbers. The mutation process is shown in Figure 3.

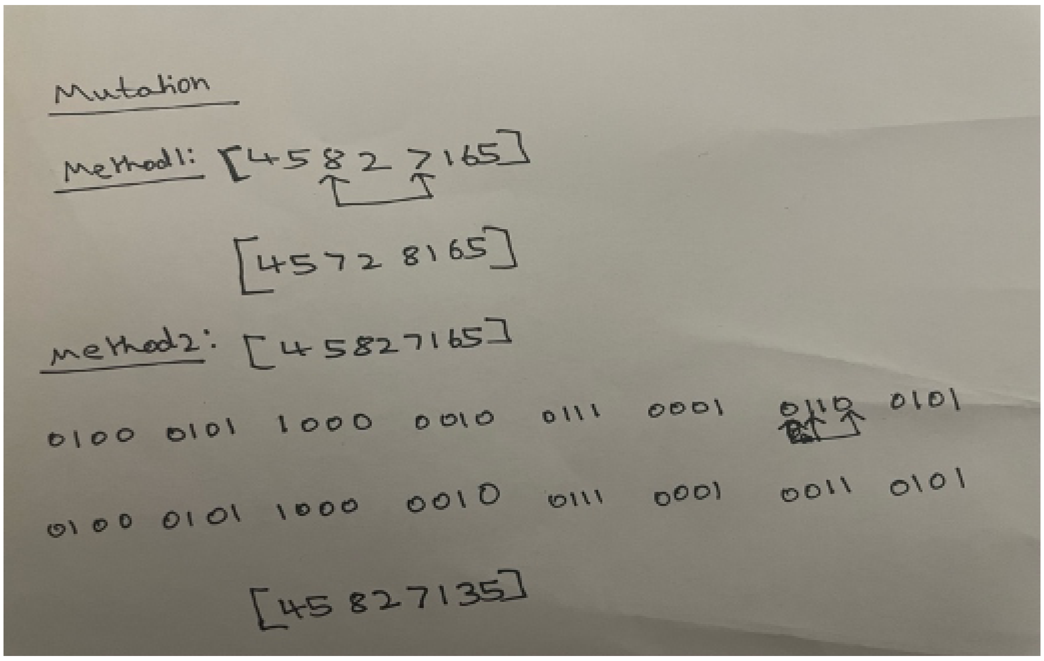
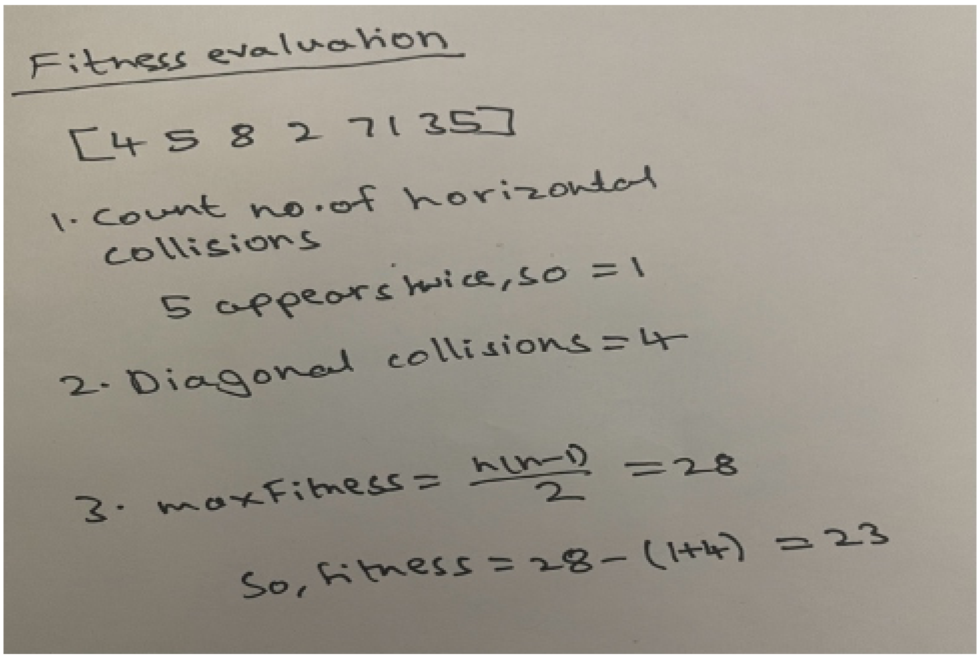


Figure 3: Mutation

6. Fitness Evaluation: Evaluate the fitness of each chromosome based on the number of conflicts in the placement. Count the number of pairs of queens that threaten each other (share the same row, column, or diagonal). The fitness value can be defined as the inverse of the number of conflicts or a similar function where lower values indicate better fitness. Figure 4 shows the Fitness Evaluation.



Fitness evaluation

[4 5 8 2 7 1 3 5]

1. count no. of horizontal collisions

    5 appeors twice, so = 1

2. Diagonal collisions = 4

3. maxFitness = $\frac{n(n-1)}{2}$ = 28

    So, fitness = 28 − (1+4) = 23

Figure 4: Fitness Evaluation

7. Termination: Repeat the selection, crossover, mutation, and fitness evaluation steps for multiple generations or until a termination criterion is met. Termination conditions could include finding a solution with no conflicts (all queens placed safely), reaching a maximum number of generations, or reaching a predefined fitness threshold.

By making these adjustments, the genetic algorithm can effectively search for solutions to the N-queens problem by iteratively improving the placement of queens on the chessboard. The general flow of the algorithm can also be seen in Figure 5 for better understanding.

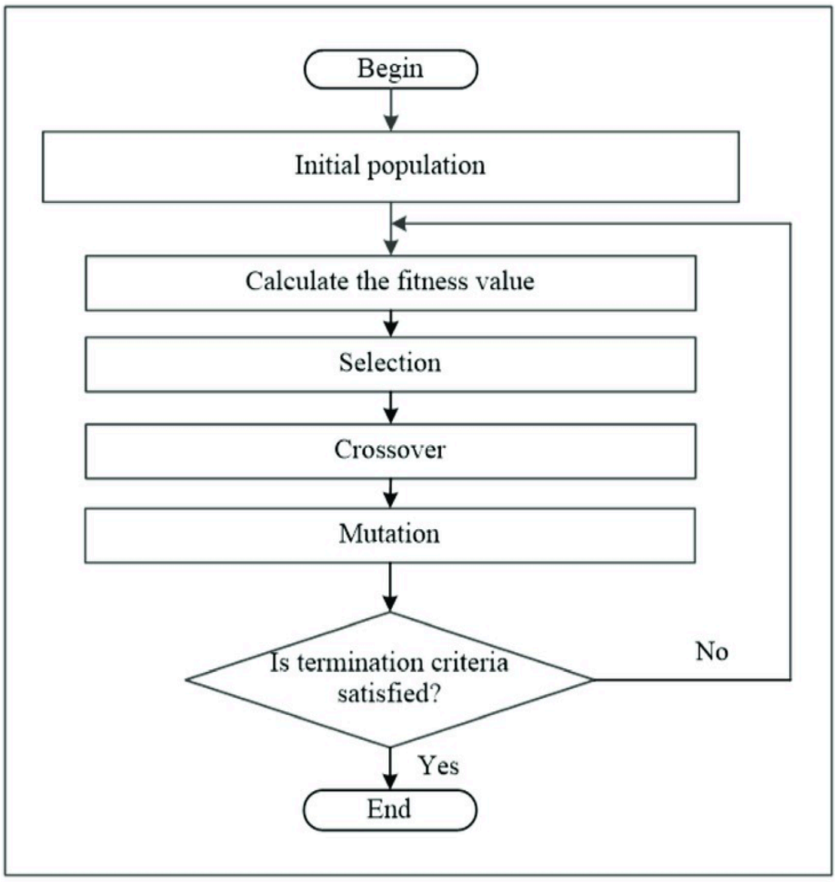

Begin
Initial population
Calculate the fitness value
Selection
Crossover
Mutation
Is termination criteria satisfied?
No
Yes
End

Figure 5: Flowchart of Genetic Algorithm

4 Implementation

Below is the full Python code that was used for the implementation of the algorithm:

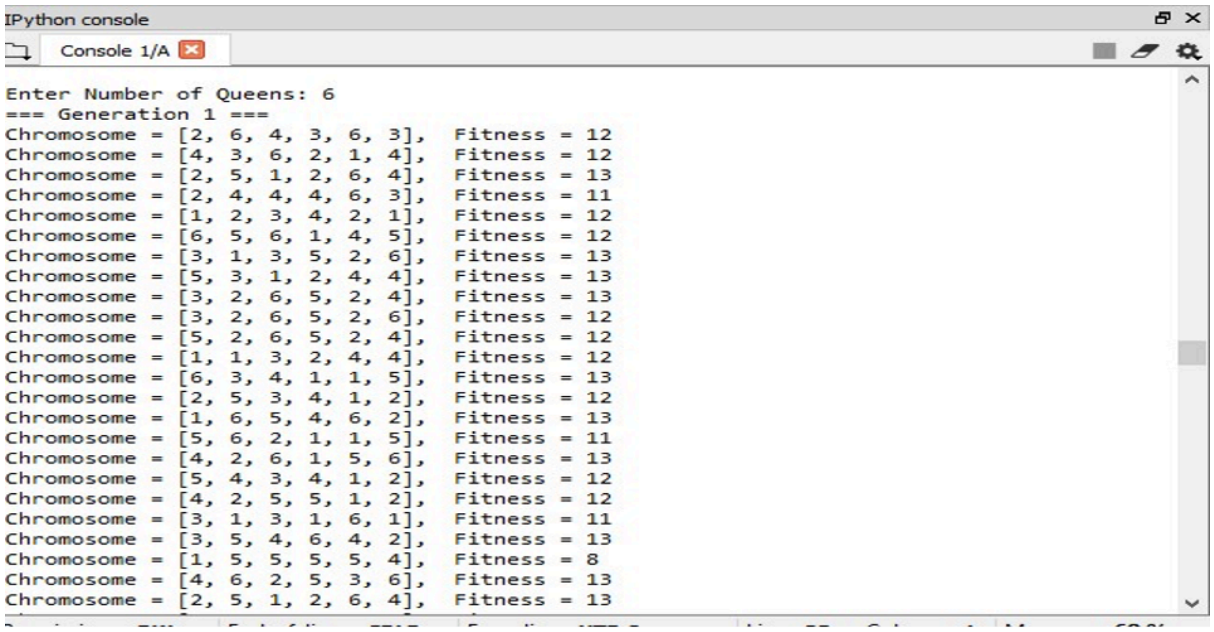
```
import random

def random_chromosome(size): #making random chromosomes
     return [ random.randint(1, nq) for _ in range(nq) ]

def fitness(chromosome):
     horizontal_collisions = sum([chromosome.count(queen)-1
for queen in chromosome])/2
     diagonal_collisions = 0

     n = len(chromosome)
     left_diagonal = [0] * 2*n
     right_diagonal = [0] * 2*n
     for i in range(n):
     left_diagonal[i + chromosome[i] - 1] += 1
     right_diagonal[len(chromosome) - i + chromosome[i] - 2]
+= 1
```

```
     diagonal_collisions = 0
     for i in range(2*n-1):
     counter = 0
     if left_diagonal[i] > 1:
          counter += left_diagonal[i]-1
     if right_diagonal[i] > 1:
          counter += right_diagonal[i]-1
     diagonal_collisions += counter / (n-abs(i-n+1))

     return int(maxFitness - (horizontal_collisions +
diagonal_collisions)) #28-(2+3)=23

def probability(chromosome, fitness):
     return fitness(chromosome) / maxFitness

def random_pick(population, probabilities):
     populationWithProbabilty = zip(population, probabilities)
     total = sum(w for c, w in populationWithProbabilty)
     r = random.uniform(0, total)
     upto = 0
     for c, w in zip(population, probabilities):
     if upto + w >= r:
          return c
     upto += w
     assert False, "Shouldn't get here"

def reproduce(x, y): #doing cross_over between two chromosomes
     n = len(x)
     c = random.randint(0, n - 1)
     return x[0:c] + y[c:n]

def mutate(x):  #randomly changing the value of a random index
of a chromosome
     n = len(x)
     c = random.randint(0, n - 1)
     m = random.randint(1, n)
     x[c] = m
     return x

def genetic_queen(population, fitness):
     mutation_probability = 0.03
     new_population = []
     probabilities = [probability(n, fitness) for n in
population]
```

```
     for i in range(len(population)):
     x = random_pick(population, probabilities) #best
chromosome 1
     y = random_pick(population, probabilities) #best
chromosome 2
     child = reproduce(x, y) #creating two new chromosomes
from the best 2 chromosomes
     if random.random() < mutation_probability:
          child = mutate(child)
     print_chromosome(child)
     new_population.append(child)
     if fitness(child) == maxFitness: break
     return new_population

def print_chromosome(chrom):
     print("Chromosome = {},  Fitness = {}"
     .format(str(chrom), fitness(chrom)))

if __name__ == "__main__":
     nq = int(input("Enter Number of Queens: ")) #say N = 8
     maxFitness = (nq*(nq-1))/2  # 8*7/2 = 28
     population = [random_chromosome(nq) for _ in range(100)]

     generation = 1

     while not maxFitness in [fitness(chrom) for chrom in
population]:
     print("=== Generation {} ===".format(generation))
     population = genetic_queen(population, fitness)
     print("")
     print("Maximum Fitness = {}".format(max([fitness(n) for n
in population])))
     generation += 1
     chrom_out = []
     print("Solved in Generation {}!".format(generation-1))
     for chrom in population:
     if fitness(chrom) == maxFitness:
          print("")
          print("One of the solutions: ")
          chrom_out = chrom
          print_chromosome(chrom)

     board = []
```

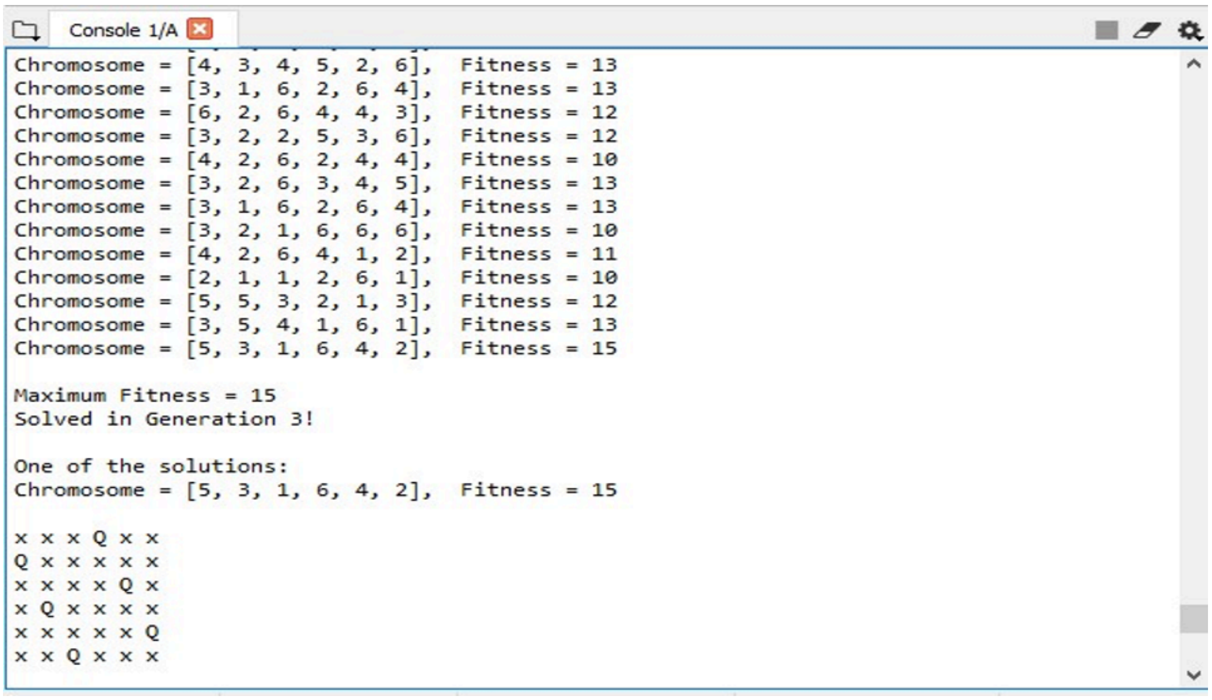
```
for x in range(nq):
board.append(["x"] * nq)

for i in range(nq):
board[nq-chrom_out[i]][i]="Q"


def print_board(board):
for row in board:
     print (" ".join(row))

print()
print_board(board)
```

5 Results



```
Enter Number of Queens: 6
=== Generation 1 ===
Chromosome = [2, 6, 4, 3, 6, 3],  Fitness = 12
Chromosome = [4, 3, 6, 2, 1, 4],  Fitness = 12
Chromosome = [2, 5, 1, 2, 6, 4],  Fitness = 13
Chromosome = [2, 4, 4, 4, 6, 3],  Fitness = 11
Chromosome = [1, 2, 3, 4, 2, 1],  Fitness = 12
Chromosome = [6, 5, 6, 1, 4, 5],  Fitness = 12
Chromosome = [3, 1, 3, 5, 2, 6],  Fitness = 13
Chromosome = [5, 3, 1, 2, 4, 4],  Fitness = 13
Chromosome = [3, 2, 6, 5, 2, 4],  Fitness = 13
Chromosome = [3, 2, 6, 5, 2, 6],  Fitness = 12
Chromosome = [5, 2, 6, 5, 2, 4],  Fitness = 12
Chromosome = [1, 1, 3, 2, 4, 4],  Fitness = 12
Chromosome = [6, 3, 4, 1, 1, 5],  Fitness = 13
Chromosome = [2, 5, 3, 4, 1, 2],  Fitness = 12
Chromosome = [1, 6, 5, 4, 6, 2],  Fitness = 13
Chromosome = [5, 6, 2, 1, 1, 5],  Fitness = 11
Chromosome = [4, 2, 6, 1, 5, 6],  Fitness = 13
Chromosome = [5, 4, 3, 4, 1, 2],  Fitness = 12
Chromosome = [4, 2, 5, 5, 1, 2],  Fitness = 12
Chromosome = [3, 1, 3, 1, 6, 1],  Fitness = 11
Chromosome = [3, 5, 4, 6, 4, 2],  Fitness = 13
Chromosome = [1, 5, 5, 5, 5, 4],  Fitness = 8
Chromosome = [4, 6, 2, 5, 3, 6],  Fitness = 13
Chromosome = [2, 5, 1, 2, 6, 4],  Fitness = 13
```

Figure 6: Result in Python Console Part 1



```
Chromosome = [4, 3, 4, 5, 2, 6],  Fitness = 13
Chromosome = [3, 1, 6, 2, 6, 4],  Fitness = 13
Chromosome = [6, 2, 6, 4, 4, 3],  Fitness = 12
Chromosome = [3, 2, 2, 5, 3, 6],  Fitness = 12
Chromosome = [4, 2, 6, 2, 4, 4],  Fitness = 10
Chromosome = [3, 2, 6, 3, 4, 5],  Fitness = 13
Chromosome = [3, 1, 6, 2, 6, 4],  Fitness = 13
Chromosome = [3, 2, 1, 6, 6, 6],  Fitness = 10
Chromosome = [4, 2, 6, 4, 1, 2],  Fitness = 11
Chromosome = [2, 1, 1, 2, 6, 1],  Fitness = 10
Chromosome = [5, 5, 3, 2, 1, 3],  Fitness = 12
Chromosome = [3, 5, 4, 1, 6, 1],  Fitness = 13
Chromosome = [5, 3, 1, 6, 4, 2],  Fitness = 15

Maximum Fitness = 15
Solved in Generation 3!

One of the solutions:
Chromosome = [5, 3, 1, 6, 4, 2],  Fitness = 15

x x x Q x x
Q x x x x x
x x x x Q x
x Q x x x x
x x x x x Q
x x Q x x x
```

Figure 7: Result in Python Console Part 2

6 Conclusion

The project concludes by presenting the results, showcasing one or more solutions to the N-Queens problem and illustrating the algorithm's efficacy in tackling complex combinatorial optimisation challenges. Overall, the project provides both a theoretical framework and a practical implementation, highlighting the power of Genetic Algorithms in addressing real-world problems with elegance and efficiency.