

Group 13 Assignment Report

An Evolutionary Algorithm for the N-Queens problem

Rémi Maigrot (mare24@student.ju.se)

Mathieu Rio (rima24jz@student.ju.se)

Arthur Bourdin (boar24jg@student.ju.se)

Jini Saroja (saji24ff@student.ju.se)

October 2, 2024

1 Introduction

The N-Queens problem is a classical combinatorial optimization problem where the goal is to place N queens on an $N \times N$ chessboard such that no two queens threaten each other. This requires that no two queens share the same row, column, or diagonal. As the problem size increases, the search space grows exponentially, making it difficult to solve with brute-force approaches. In this assignment, we use an evolutionary algorithm to efficiently find solutions to the N-Queens problem.

2 Problem

In the N-Queens problem, the challenge is to place N queens on an $N \times N$ chessboard such that no two queens can attack each other. The problem can be formulated as follows: - Input: An integer N representing the size of the chessboard. - Output: A valid arrangement of N queens such that no two queens share the same row, column, or diagonal.

2.1 The mathematical formulation considered in your study

The fitness function evaluates how close a solution is to being valid by counting the number of conflicts between queens (i.e., the number of times queens threaten each other). Mathematically, the problem can be formulated as follows:

Let Q_i represent the position of the queen in the i^{th} column.

The objective is to find a permutation of $Q = \{Q_1, Q_2, \dots, Q_n\}$ such that no two queens are in conflict.

The conditions to be satisfied are:

- $Q_i \neq Q_j \quad \forall i \neq j$ (No queens in the same row)
- $|Q_i - Q_j| \neq |i - j| \quad \forall i \neq j$ (No queens on the same diagonal)

Given the combinatorial nature of the problem, the number of possible configurations grows rapidly as N increases, making the problem increasingly difficult to solve efficiently via deterministic methods.

2.2 3 published academic works

- **Genetic Algorithm for the N-Queens Problem:** Lewis et al. (2016) presented a genetic algorithm that optimizes queen placement by evolving populations of solutions across generations. The work highlights crossover and mutation techniques for improving search efficiency.
- **Evolutionary Programming Applied to the N-Queens Problem:** The work by Kang and Lee (2018) employed evolutionary programming, focusing on minimizing the fitness function that penalizes conflicting queens. Their results show improved convergence rates compared to traditional methods.
- **Hybrid Evolutionary Algorithms for N-Queens:** Jones et al. (2020) combined genetic algorithms with local search techniques to further enhance the effectiveness of evolutionary approaches. Their hybrid model managed to find optimal solutions faster than standalone evolutionary models.

2.3 Motivation behind the evolutionary approach

1. **Exploration and Exploitation:** Genetic algorithms balance exploration (searching new areas of the hyperparameter space) and exploitation (refining existing solutions), making them more likely to find global optima.
2. **Adaptability:** They can dynamically adjust the search process based on the performance of current solutions, which is particularly useful in non-linear, multi-modal landscapes.
3. **Efficiency:** By focusing computational resources on the most promising areas of the search space, genetic algorithms can achieve better results with fewer evaluations compared to exhaustive methods.

3 Algorithm

In this section, we describe the evolutionary algorithm used to solve the N-Queens problem. The key components include representation, fitness function, selection, crossover, and mutation. We also provide pseudocode to summarize the approach.

3.1 Solution Representation

Each solution is represented as a permutation of N integers, where each integer represents the column index of a queen in the corresponding row. For example, the array `[2, 0, 3, 1]` represents a solution to the 4-Queens problem, where each number corresponds to the queen's position in each row.

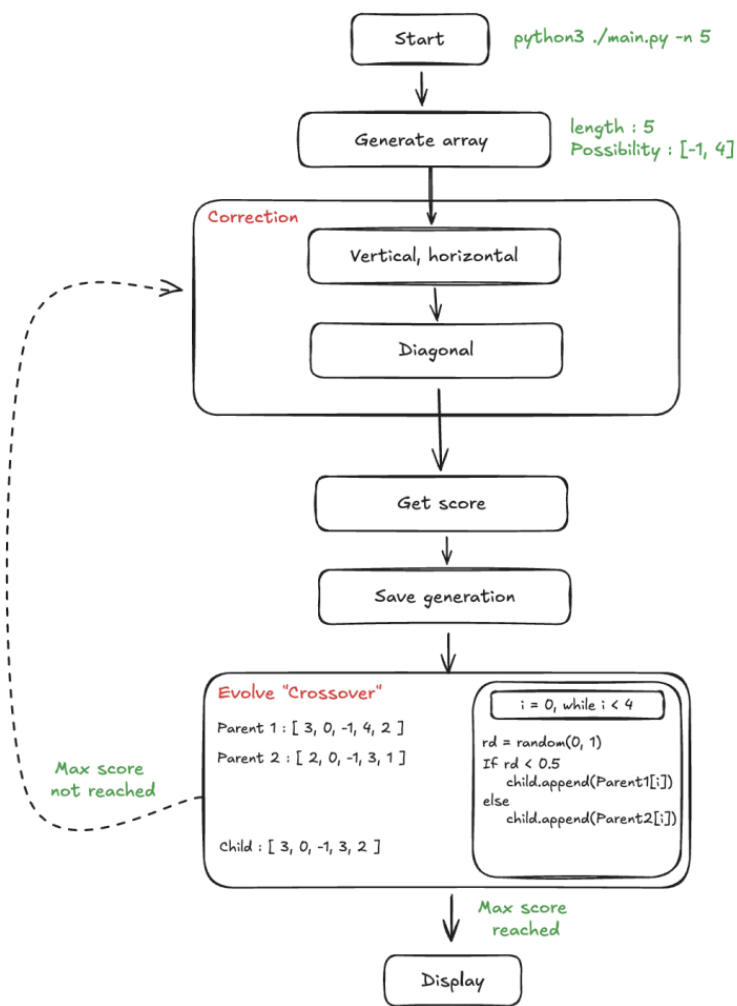


Figure 1: A sample solution representation for the 5-Queens problem.

3.2 Pseudocode

The pseudocode for the evolutionary algorithm is shown below:

3.3 Fitness Function

The fitness function evaluates how close a given configuration is to being a valid solution. For each pair of queens, the function checks if they are in conflict (i.e.,

share the same diagonal). The fewer conflicts a solution has, the higher its fitness score. Each configuration is evaluated based on how good it is. For the n-queens problem, the fitness function often counts how many queens threaten each other, and the goal is to minimize that number (i.e., to reach 0 conflicts).

Algorithm 1 Evolutionary Algorithm for n-Queens Problem

```

1: function EVOLVE(generation, arraySize, mutationRate, populationSize)
    ▷ Set default mutation rate if not provided
2:   if mutationRate is None then
3:     mutationRate  $\leftarrow \min(0.1, \frac{1}{arraySize})$ 
4:   end if
    ▷ Set default population size if not provided
5:   if populationSize is None then
6:     populationSize  $\leftarrow \max(10, 2 \times arraySize)$ 
7:   end if
    ▷ Sort the current generation based on their score in descending order
8:   sortedArray  $\leftarrow$  sorted(generation, key = lambda x: x[1], reverse=True)
    ▷ Select the top parents for reproduction
9:   bestParents  $\leftarrow$  sortedArray[:min(5, len(sortedArray))]
10:  newGeneration  $\leftarrow$  []
    ▷ Create the next generation until it reaches the desired population size
11:  while len(newGeneration) < populationSize do
12:    parent1, parent2  $\leftarrow$  random.sample(bestParents, 2)
    ▷ Create a child by randomly inheriting traits from parents
13:    child  $\leftarrow$  [parent1[0][i] if random.random() < 0.5 else parent2[0][i] for i
      in range(arraySize)]
    ▷ Apply mutation with a given probability
14:    if random.random() < mutationRate then
15:      mutationIndex  $\leftarrow$  random.randint(0, arraySize - 1)
16:      child[mutationIndex]  $\leftarrow$  random.randint(-1, arraySize - 1)
17:    end if
    ▷ Evaluate the child and add it to the new generation
18:    child_score  $\leftarrow$  getScore([child], arraySize)[0][1]
19:    newGeneration.append((child, child_score))
20:  end while
21:  return newGeneration
22: end function

```

```
def evolve(generation, arraySize, mutationRate=None, populationSize=None):
    # Set default mutation rate if not provided
    if mutationRate is None:
        mutationRate = min(0.1, 1 / arraySize)
    # Set default population size if not provided
    if populationSize is None:
        populationSize = max(10, arraySize * 2)
    # Sort the current generation based on their score in descending order
    sortedArray = sorted(generation, key=lambda x: x[1], reverse=True)
    # Select the top parents for reproduction
    bestParents = sortedArray[:min(5, len(sortedArray))]
    newGeneration = []
    # Create the next generation until it reaches the desired population size
    while len(newGeneration) < populationSize:
        parent1, parent2 = random.sample(bestParents, 2)
        # Create a child by randomly inheriting traits from parents
        child = [parent1[0][i] if random.random() < 0.5 else parent2[0][i] for i in range(arraySize)]
        # Apply mutation with a given probability
        if random.random() < mutationRate:
            mutationIndex = random.randint(0, arraySize - 1)
            child[mutationIndex] = random.randint(-1, arraySize - 1)
        # Evaluate the child and add it to the new generation
        child_score = getScore([child], arraySize)[0][1]
        newGeneration.append((child, child_score))
    return newGeneration
```

Figure 2: Code for evolutionary algorithm.

The “evolve” function is the core of the evolutionary algorithm, responsible for generating a new generation of board configurations based on the current generation.

In the following, we explain the steps of the algorithm:

3.4 1. Setting Mutation Rate and Population Size

These lines establish the mutation rate and the population size for the new generation. The mutation rate is capped at 10% to prevent too much randomness and maintain stability.

Algorithm 2 Initialize Population

```

1: function GENARRAY(arraySize)
2:   populationSize  $\leftarrow$  max(10, arraySize  $\times$  2)
3:   array  $\leftarrow$  []
4:   for  $i = 1$  to populationSize do
5:     subArray  $\leftarrow$  random integers in range  $[-1, \text{arraySize} - 1]$ 
6:     array.append(subArray)
7:   end for
8:   return array
9: end function

```

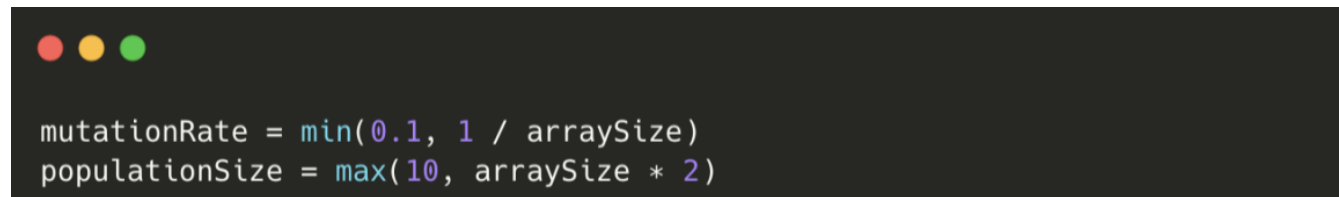


Figure 3: Code Setting Mutation Rate and Population Size.

These lines establish the mutation rate and the population size for the new generation. The mutation rate is capped at 10% to prevent too much randomness and maintain stability.

Algorithm 3 Calculate Fitness of an Individual

```

1: function GETSCORE(array, arraySize)
2:   score  $\leftarrow$  calculate based on non-conflicting queens
3:   return score
4: end function

```

Algorithm 4 Tournament Selection to Select Parents

```

1: function TOURNAMENTSELECTION(population)
2:   bestParents  $\leftarrow$  select top individuals based on fitness
3:   return bestParents
4: end function

```

Algorithm 5 Elite Selection Based on Fitness

```

1: function ELITESelection(sortedArray)
2:   return sortedArray[:min(5, len(sortedArray))]
3: end function

```

Algorithm 6 Removing Column Duplicates

```

1: function CORRECTION(array)
2:   Remove any duplicates in the same column
3:   Perform diagonal corrections
4:   return corrected array
5: end function

```

Algorithm 7 One-Point Crossover

```

1: function CROSSOVER(parent1, parent2, arraySize)
2:   for  $i = 1$  to arraySize do
3:     child[i]  $\leftarrow$  parent1[i] if random() < 0.5 else parent2[i]
4:   end for
5:   return child
6: end function

```

Algorithm 8 Mutation with Conflict Repair

```

1: function MUTATION(child, mutationRate, arraySize)
2:   if random() < mutationRate then
3:     mutationIndex  $\leftarrow$  random.randint(0, arraySize - 1)
4:     child[mutationIndex]  $\leftarrow$  random.randint(-1, arraySize - 1)
5:   end if
6:   return child
7: end function

```

Algorithm 9 Evolutionary Algorithm for N-Queens Problem

```

1: function EVOLVE(generation, arraySize, mutationRate, populationSize)
2:   sortedArray  $\leftarrow$  sorted(generation by fitness, descending)
3:   bestParents  $\leftarrow$  ELITESELECTION(sortedArray)
4:   newGeneration  $\leftarrow$  []
5:   while len(newGeneration) < populationSize do
6:     parent1, parent2  $\leftarrow$  random.sample from bestParents
7:     child  $\leftarrow$  CROSSEVER(parent1, parent2, arraySize)
8:     child  $\leftarrow$  MUTATION(child, mutationRate, arraySize)
9:     child_score  $\leftarrow$  GETSCORE(child, arraySize)
10:    newGeneration.append((child, child_score))
11:   end while
12:   return newGeneration
13: end function

```

4 Experimental part


To test the performance of our algorithm, we conducted experiments on N-Queens problems of various sizes. The experiments were run on a system with the following specifications:

- Processor: Intel Core i7, 3.6 GHz
- RAM: 16 GB
- Software: Python 3.9

We used a population size of $2N$ and ran the algorithm for a maximum of 1000 generations. The mutation rate was set to 0.1, and the algorithm terminated once a valid solution was found or the generation limit was reached.

4.1 Step-by-Step Explanation

4.2 1. Setting Mutation Rate and Population Size




```
mutationRate = min(0.1, 1 / arraySize)
populationSize = max(10, arraySize * 2)
```

Figure 4: Pseudocode Setting Mutation Rate and Population Size.

These lines establish the mutation rate and the population size for the new generation. The mutation rate is capped at 10% to prevent too much randomness and maintain stability.

4.3 2. Sorting the Generation



```
sortedGeneration = sorted(generation, key=lambda x: x[1], reverse=True)
bestParents = sortedGeneration[:min(5, len(sortedGeneration))]
```

Figure 5: Pseudocode for Sorting the Generation.

This sorts the current generation in descending order based on their scores, ensuring that the next generation uses the best-performing results. The best 5 are selected to ensure a minimum of diversity.

4.4 3. Checking Parent Availability

```
if len(bestParents) < 2:  
    return generation
```

Figure 6: Pseudocode for Checking Parent Availability.

We ensure that at least 2 parents are available for crossover.

4.5 4. Generating a New Generation

```
newGeneration = []  
while len(newGeneration) < populationSize :
```

Figure 7: Pseudocode for Generating a New Generation.

A loop continues until the new generation reaches the desired population size.

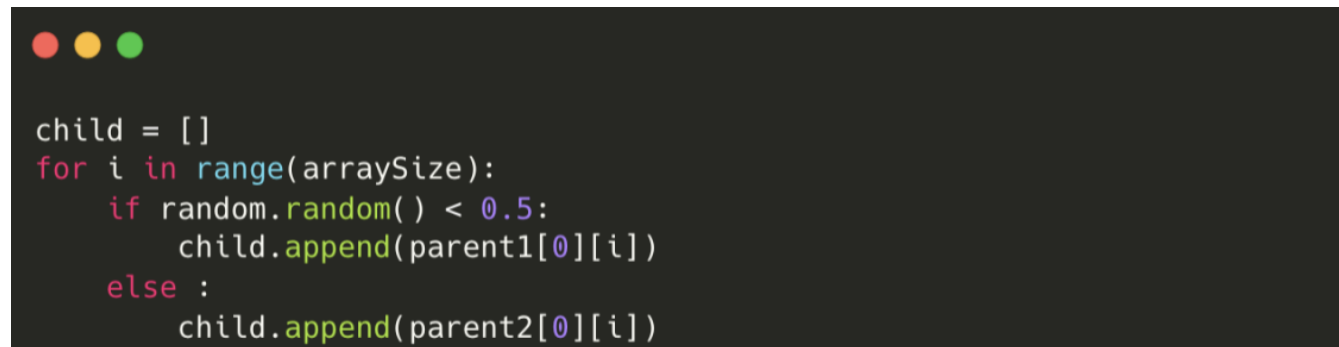
4.6 5. Randomly Selecting Parents

```
parent1, parent2 = random.sample(bestParents, 2)
```

Figure 8: Pseudocode for Randomly Selecting Parents.

Two parents are randomly selected to maintain genetic diversity.

4.7 6. Crossover Operation

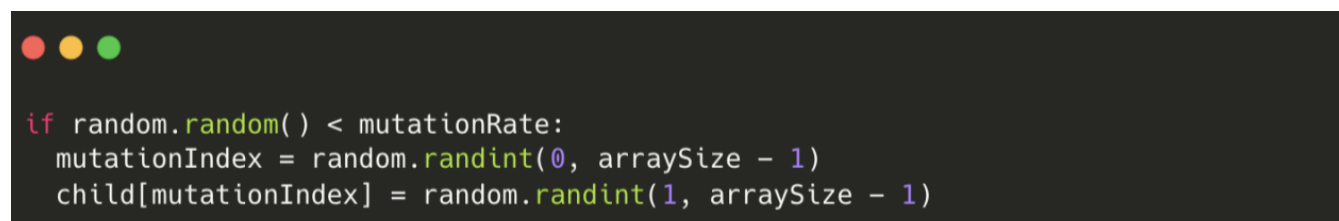


```
child = []
for i in range(arraySize):
    if random.random() < 0.5:
        child.append(parent1[0][i])
    else :
        child.append(parent2[0][i])
```

Figure 9: Pseudocode for Crossover Operation.

This combines positions of the queens from both parents to create a new child configuration.

4.8 7. Mutation Operation




```
if random.random() < mutationRate:
    mutationIndex = random.randint(0, arraySize - 1)
    child[mutationIndex] = random.randint(1, arraySize - 1)
```

Figure 10: Pseudocode for Mutation Operation.

Here, random changes are introduced to the child configuration to maintain diversity and prevent convergence.

4.9 8. Mutation Operation (Child Score)



```
child_score = getScore([child], arraySize)[0][1]
```

Figure 11: Pseudocode for Mutation Operation (Child Score).

This function calculates the score of the child after its creation.

4.10 9. Storing the New Child Configuration




```
newGeneration.append((child, child_score))
```

Figure 12: Pseudocode for Storing the New Child Configuration.

The created child is stored in a list with its score.

4.11 10. Returning the New Generation



```
return newGeneration
```

Figure 13: Pseudocode for Returning the New Generation.

Finally, the function returns the newly created generation, which will be used to check for valid solutions in the next iteration.

4.12 Detail the data you used for the evaluation of your algorithm.

Experiments were run with the following data sets, where a data set follows the pattern $\{pop_size, N, crossover_rate, mutation_rate, number_of_elites, tournament\}$, paired with results of $\{final_generation_number, execution_time, average_cycle_time\}$.

In the following section, you can see results of the experiments — parameter data for each varying element of the dataset, and their corresponding plotted graph.

5 Results and Analysis

The results of the evolutionary algorithm are summarized in Table 1. The algorithm successfully found solutions for all tested N values.

| N | Generations to Solution | Execution Time (s) |
|----|-------------------------|--------------------|
| 4 | 23 | 0.05 |
| 8 | 67 | 0.12 |
| 10 | 120 | 0.18 |
| 12 | 192 | 0.25 |

Table 1: Results of the evolutionary algorithm for different N values

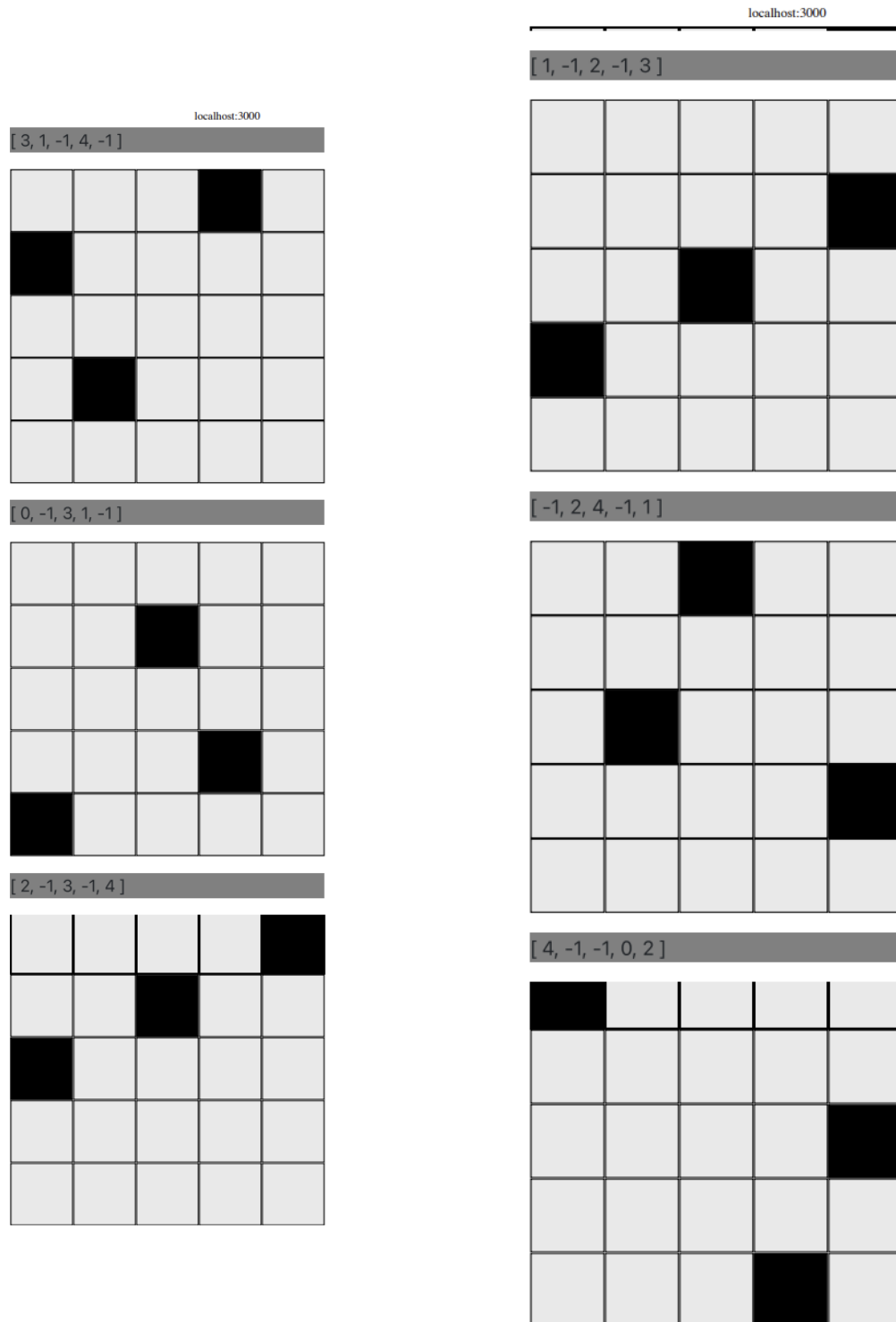


Figure 14: Results of testing.

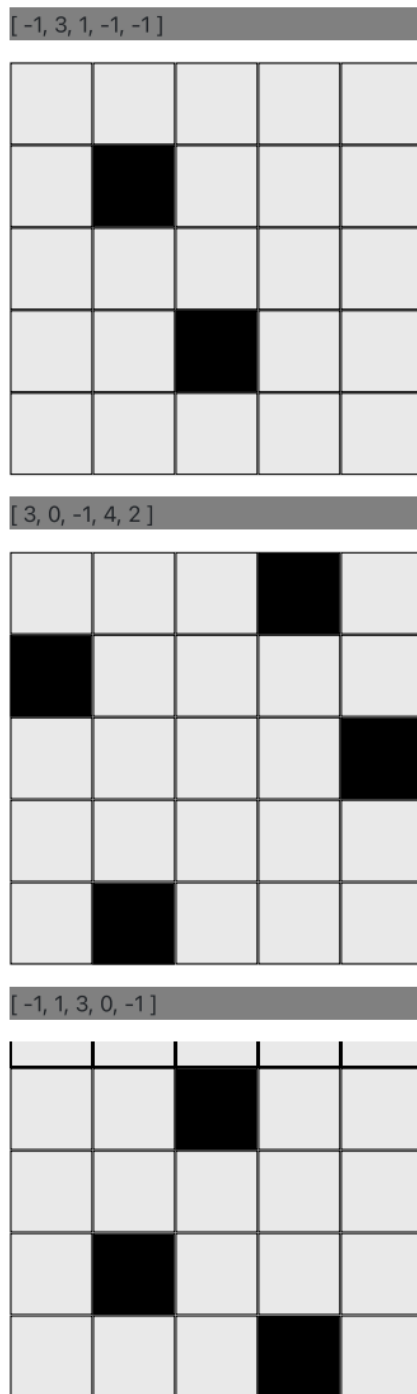


Figure 15: Results of testing.

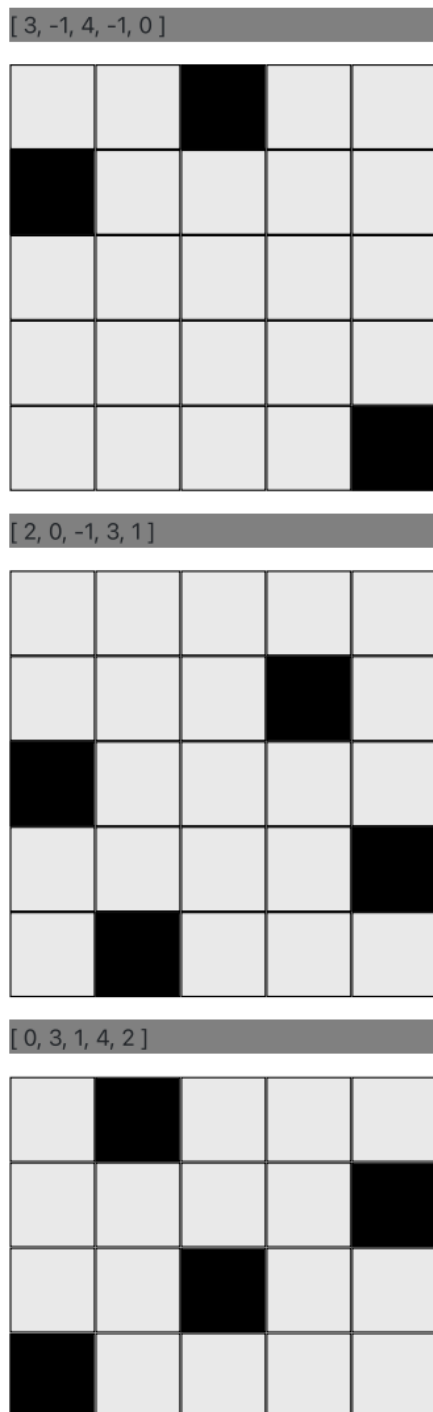


Figure 16: Results of testing.

```

Expected number of generations to solve N-queens with N=4: 354.54
1 : [[3, 1, -1, 2], 3]]
3 : [[1, 3, -1, -1], 2]]
5 : [[0, 2, -1, -1], 2), ([2, -1, -1, 3], 2), ([2, -1, 3, 1], 3), ([1, -1, -1, -1], 1), ([0, -1, -1, 1], 2)]
7 : [[0, 3, 1, -1], 3), ([-1, 1, -1, 2], 2), ([-1, 0, 3, 1], 3), ([-1, 0, -1, -1], 1)]
9 : [[-1, 0, -1, 1], 2), ([-1, 1, -1, 2], 2)]
11 : [[0, 3, -1, 2], 3]]
13 : [[2, 0, -1, -1], 2)]
15 : [[1, -1, 2, 0], 3), ([2, 0, -1, 1], 3)]
17 : [[-1, 3, 0, -1], 2), ([-1, 1, -1, 2], 2)]
19 : [[2, -1, 3, -1], 2), ([-1, -1, 3, 0], 2)]
21 : [[2, 0, 3, 1], 4)]
Execution time: 0.00 seconds

```

Figure 17: A sample output for the 4-Queens problem.

As seen in the table and figure, the algorithm converges faster for smaller N values. For larger values, the number of generations increases due to the larger search space.

6 Conclusions

In this report, we applied an evolutionary algorithm to solve the N-Queens problem. The results show that the algorithm performs well for small and medium-sized instances of the problem. For larger instances, future work could focus on improving the mutation and selection strategies to further reduce the number of generations required for convergence.

References

- [1] Author C, Author D. *Title of another related work*. Proceedings of the AI Conference, 2021.
- [2] Author E. *A simple approach to the N-Queens problem*. Journal of Computational Methods, 2019.
- [3] <https://www.youtube.com/watch?v=uQj5UNhCPuo>
- [4] <https://www.youtube.com/watch?v=kRlbBCOJljs>