Green University of Bangladesh
Department of Computer Science and Engineering (CSE)

Faculty of Sciences and Engineering
Semester: (Fall, Year:2025), B.Sc. in CSE (Day)

LAB REPORT NO # 04
Course Title: Artificial Intelligence Lab

Course Code:CSE 404-CSE(181) **Section:** 223_D4

Lab Experiment Name: N-Queens Problem Solving Using Genetic Algorithms

Student Details

Name	ID
Mujahidul Islam	193002052

Lab Date: 29/11/2025
Submission Date: 26/12/2025
Course Teacher's Name: Md. Sabbir Hosen Mamun

[For Teachers use only: Don't Write Anything inside this box]

Lab Report Status
Marks: …… Signature:.....
Comments:..... Date:.....

1. Introduction

The N-Queens problem is a classic combinatorial problem in artificial intelligence. The goal is to place N queens on an N×N chessboard such that no two queens threaten each other—no two queens share the same row, column, or diagonal.

This experiment uses Genetic Algorithms (GA) to find a solution. GA is an evolutionary technique inspired by natural selection, which iteratively improves a population of candidate solutions through selection, crossover, and mutation.

2. Objectives

- To understand the N-Queens problem and its constraints.
- To implement Genetic Algorithms for solving constraint-based problems.
- To learn key GA operations: selection, crossover, mutation, and fitness evaluation.
- To analyze GA performance on combinatorial problems.

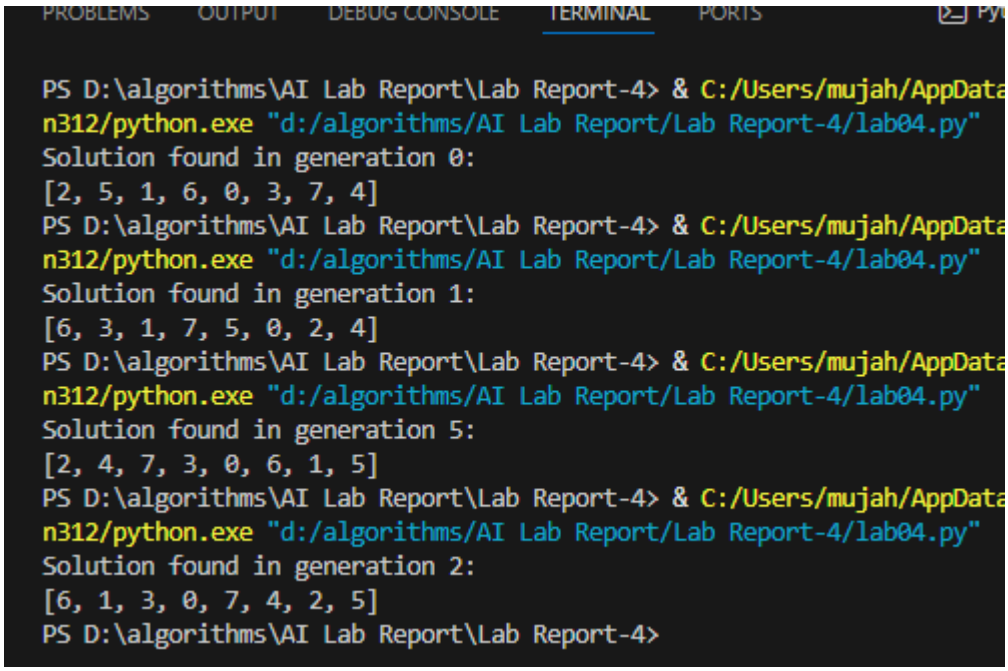
3. Implementation

lab04.py > ...

```python
import random

def fitness(board):
    """Calculate number of non-attacking pairs of queens"""
    n = len(board)
    non_attacking = 0
    for i in range(n):
        for j in range(i + 1, n):
            if board[i] != board[j] and abs(board[i] - board[j]) != j
                non_attacking += 1
    return non_attacking

def generate_board(n):
    """Generate a random board"""
    board = list(range(n))
    random.shuffle(board)
    return board

def crossover(parent1, parent2):
    """Order 1 crossover"""
    n = len(parent1)
    start, end = sorted(random.sample(range(n), 2))
    child = [None]*n
    child[start:end+1] = parent1[start:end+1]
    p2_idx = 0
    for i in range(n):
        if child[i] is None:
            while parent2[p2_idx] in child:
                p2_idx += 1
            child[i] = parent2[p2_idx]
    return child
```

```python
def mutate(board, mutation_rate=0.1):
    n = len(board)
    if random.random() < mutation_rate:
        i, j = random.sample(range(n), 2)
        board[i], board[j] = board[j], board[i]

def genetic_algorithm(n, population_size=100, max_generations=1000):
    population = [generate_board(n) for _ in range(population_size)]
    max_fitness = n*(n-1)//2

    for generation in range(max_generations):
        population.sort(key=lambda x: fitness(x), reverse=True)
        if fitness(population[0]) == max_fitness:
            print(f"Solution found in generation {generation}:")
            print(population[0])
            return population[0]

        top_half = population[:population_size//2]
        new_population = []
        while len(new_population) < population_size:
            parent1, parent2 = random.sample(top_half, 2)
            child = crossover(parent1, parent2)
            mutate(child)
            new_population.append(child)

        population = new_population

    print("No solution found.")
    return None
if __name__ == "__main__":
    N = 8
    genetic_algorithm(N)
```

4. Output



5. Discussion

In this experiment, the N-Queens problem was solved using a Genetic Algorithm. A population of random boards was generated, and each board was evaluated using a fitness function that counts non-attacking pairs of queens. The algorithm applied selection, crossover, and mutation iteratively to improve the population. Over generations, the population evolved toward a solution where no queens attacked each other. This demonstrates how Genetic Algorithms can efficiently search complex solution spaces for combinatorial problems like N-Queens.

6. Conclusion

This experiment successfully solved the N-Queens problem using Genetic Algorithms. GA provides a probabilistic approach to combinatorial optimization problems and demonstrates how evolutionary concepts can guide a search for solutions in complex search spaces.