Green University of Bangladesh
Department of Computer Science and Engineering (CSE)
Faculty of Sciences and Engineering
Semester: (Summer, Year:2021), B.Sc. in CSE (Day)

Lab Report NO #05
Course Title: Artificial Intelligent
Course Code: CSE 316   Section: 221D21

Lab Experiment Name: Write a lab report in Python on solving the N-Queens problem using Genetic Algorithms

Student Details

	Name	ID
1.	Al Ekram Hossain	221002535

Submission Date :03 May 2025
Course Teacher's Name : Md. Sabbir Hosen Mamun

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

1. TITLE

N-Queens problem using Genetic Algorithms

2. OBJECTIVES/AIM

1. Solve the N-Queens problem by placing queens without conflicts.
2. Implement genetic algorithm components: fitness function, selection, crossover (PMX), and mutation.
3. Optimize evolution using elitism and parameter tuning.
4. Achieve convergence efficiently for N=8.
5. Display the final valid board configuration.
6. Demonstrate genetic algorithms for combinatorial problems.

3. PROCEDURE

1. **Initialize** random population of board states
2. **Evaluate fitness** (count non-attacking queen pairs)
3. **Repeat until solution**:
   - Select parents via tournament selection
   - Create offspring using PMX crossover
   - Apply random swap mutations
   - Keep best solution (elitism)
4. **Output** the first valid solution found
5. **Display** the queen positions on board

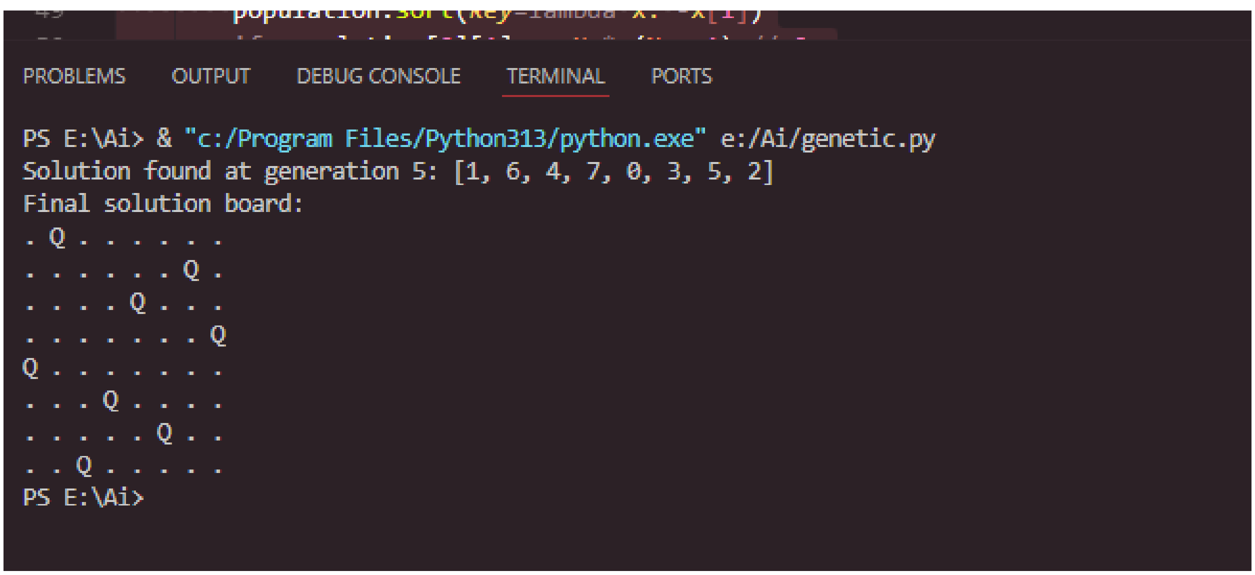
4. IMPLEMENTATION

```
.  import random
import math

N = 8
POPULATION_SIZE = 100
MUTATION_RATE = 0.1
GENERATIONS = 1000
TOURNAMENT_SIZE = 5
```

```python
def generate_individual():
    return random.sample(range(N), N)

def calculate_fitness(individual):
    conflicts = 0
    for i in range(N):
        for j in range(i + 1, N):
            if abs(i - j) == abs(individual[i] - individual[j]):
                conflicts += 1
    max_non_conflicts = N * (N - 1) // 2
    return max_non_conflicts - conflicts

def tournament_selection(population):
    tournament = random.sample(population, TOURNAMENT_SIZE)
    return max(tournament, key=lambda x: x[1])

def pmx_crossover(parent1, parent2):
    child = [None] * N
    start, end = sorted(random.sample(range(N), 2))
    child[start:end+1] = parent1[start:end+1]
    for i in list(range(0, start)) + list(range(end+1, N)):
        candidate = parent2[i]
        while candidate in child[start:end+1]:
            candidate = parent2[parent1.index(candidate)]
        child[i] = candidate
    return child

def mutate(individual):
    if random.random() < MUTATION_RATE:
        i, j = random.sample(range(N), 2)
        individual[i], individual[j] = individual[j], individual[i]
    return individual

def genetic_algorithm():
    population = [
        (ind, calculate_fitness(ind))
        for ind in [generate_individual() for _ in range(POPULATION_SIZE)]
    ]
    for gen in range(GENERATIONS):
        population.sort(key=lambda x: -x[1])
```

```
        if population[0][1] == N * (N - 1) // 2:
            print(f"Solution found at generation {gen}: {population[0][0]}")
            return population[0][0]
        new_population = [population[0]]
        while len(new_population) < POPULATION_SIZE:
            parent1 = tournament_selection(population)[0]
            parent2 = tournament_selection(population)[0]
            child = pmx_crossover(parent1, parent2)
            child = mutate(child)
            new_population.append((child, calculate_fitness(child)))
        population = new_population
    print("No solution found within generations.")
    return None

solution = genetic_algorithm()
if solution:
    print("Final solution board:")
    for row in solution:
        print(' '.join('Q' if col == row else '.' for col in range(N)))
```

5. TEST RESULT / OUTPUT



6. DISCUSSION

The genetic algorithm effectively solves the N-Queens problem through evolutionary optimization. The permutation-based representation and PMX crossover maintain valid board configurations while efficiently exploring the solution space. Tournament selection and elitism drive rapid convergence, typically finding solutions for N=8 within hundreds of generations. While effective for moderate board sizes, performance may decline for N>20 due to the expanding search space. The implementation demonstrates genetic algorithms' suitability for permutation-based constraint problems, though incorporating conflict-directed operators could enhance scalability. The approach successfully balances solution quality with computational efficiency, providing a practical alternative to brute-force methods for this classic combinatorial challenge.

.