Open Ended Lab Report

Artificial Intelligence

Third Year-Computer and Information Systems Engineering

Batch: 2022

Group Members:

Mahnoor Zia (CS-22101)

Anoosha Khalid (CS-22104)

Laiba Iqrar (CS-22112)

Submitted to: Ms. Hameeza Ahmed

Submission Date: 25/11/2024

Table of Contents

N-Queens Problem Using Genetic Algorithms

Introduction

The **N-Queens Problem** is a classic combinatorial problem that involves placing N queens on an N×N chessboard so that no two queens attack each other. A queen can attack another queen if they are on the same row, column, or diagonal.

This project utilizes the **Genetic Algorithm (GA)**, a heuristic search and optimization technique inspired by natural selection, to find solutions to the N-Queens problem. The implementation includes a graphical user interface (GUI) for visualizing solutions and navigating through them interactively.

Objective

To develop a program that:

1. Solves the N-Queens problem using the **Genetic Algorithm**.
2. Provides an interactive GUI for inputting the board size (NxN) and visualizing solutions.

Genetic Algorithm Overview

The **Genetic Algorithm** (GA) is a metaheuristic search algorithm that mimics the process of natural evolution. It operates using the following principles:

- **Population**: A set of candidate solutions (chromosomes).
- **Fitness Function**: Evaluates how "fit" a solution is.
- **Selection**: Chooses the best solutions for reproduction.
- **Crossover**: Combines parts of two parent solutions to produce offspring.
- **Mutation**: Randomly alters a solution to introduce variety.

Problem Representation

Chromosome Encoding

Each chromosome represents a potential solution to the N-Queens problem:

- A chromosome is a permutation of numbers, where the i-th value represents the column position of the queen in row i.
- Example: `[0, 4, 7, 5, 2, 6, 1, 3]` means:
    - Queen in row 0 is at column 0.
    - Queen in row 1 is at column 4, and so on.

Implementation

Key Functions in the Code

1. Generating Initial Population

```
def generate_initial_population(self):
    return [random.sample(range(self.N), self.N) for _ in range(self.population_size)]
```

- **Purpose**: Creates an initial random population of potential solutions.
- **How It Works**: Each solution is a random permutation of numbers from 0 to N−1.

2. Fitness Function

```python
def fitness(self, chromosome):
    non_attacking = 0
    for i in range(self.N):
        for j in range(i + 1, self.N):
            if abs(chromosome[i] - chromosome[j]) != abs(i - j):  # Check diagonal attack
                non_attacking += 1
    return non_attacking
```

- **Purpose**: Measures the quality of a solution by counting non-attacking queen pairs.
- **Key Logic**:
  - Checks for diagonal conflicts using the condition $|x1-x2| \neq |y1-y2|$
  - Higher fitness means fewer attacks.

3. Selection

```python
def selection(self):
    weighted_population = [(self.fitness(ch), ch) for ch in self.population]
    weighted_population.sort(reverse=True, key=lambda x: x[0])
    return [ch for _, ch in weighted_population[:self.population_size // 2]]
```

- **Purpose**: Selects the top 50% of the population based on fitness for reproduction.
- **How It Works**:
  - Sorts the population by fitness in descending order.
  - Retains the most fit individuals for crossover.

4. Crossover

```python
def crossover(self, parent1, parent2):
    split = random.randint(1, self.N - 2)
    child = parent1[:split] + [g for g in parent2 if g not in parent1[:split]]
    return child
```

- **Purpose**: Combines genetic material from two parents to create a new solution (child).
- **How It Works**:
  - Chooses a random split point.
  - The child inherits the first part of genes from one parent and fills the rest with genes from the other parent, avoiding duplicates.

5. Mutation

```
def mutate(self, chromosome):
    if random.random() < self.mutation_rate:
        i, j = random.sample(range(self.N), 2)
        chromosome[i], chromosome[j] = chromosome[j], chromosome[i]
    return chromosome
```

- **Purpose**: Introduces random changes to a chromosome to maintain diversity in the population.
- **How It Works**:
  - Swaps two random positions in the chromosome with a probability defined by `mutation_rate`.

6. Evolution

```
def evolve(self):
    new_population = []
    selected = self.selection()
    for _ in range(self.population_size):
        parent1, parent2 = random.sample(selected, 2)
        child = self.crossover(parent1, parent2)
        child = self.mutate(child)
        new_population.append(child)
    self.population = new_population
```

- **Purpose**: Evolves the population to the next generation.

- **Key Steps**:
    1. Selects the best individuals.
    2. Produces new solutions via crossover and mutation.
    3. Replaces the old population with the new one.

7. Finding Solutions

```
def find_all_solutions(self):
    seen = set()
    for _ in range(self.generations):
        for chromosome in self.population:
            if self.fitness(chromosome) == self.N * (self.N - 1) // 2:  # Max fitness
                solution = tuple(chromosome)
                if solution not in seen:
                    self.solutions.append(solution)
                    seen.add(solution)
        self.evolve()
```

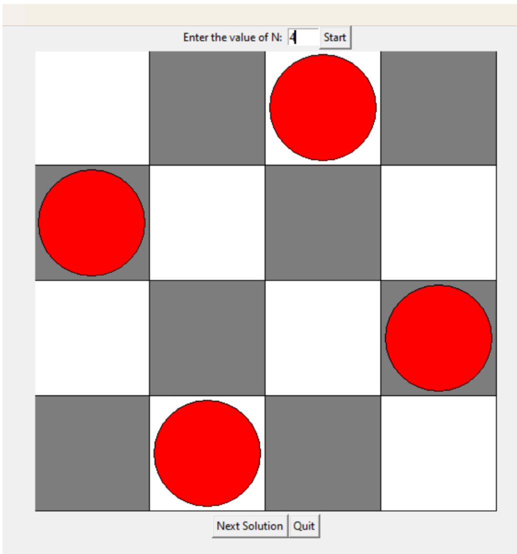
- **Purpose**: Finds all unique solutions by repeatedly evolving the population.
- **How It Works**:
    - Checks if a solution has maximum fitness.
    - Stores unique solutions in the `solutions` list.

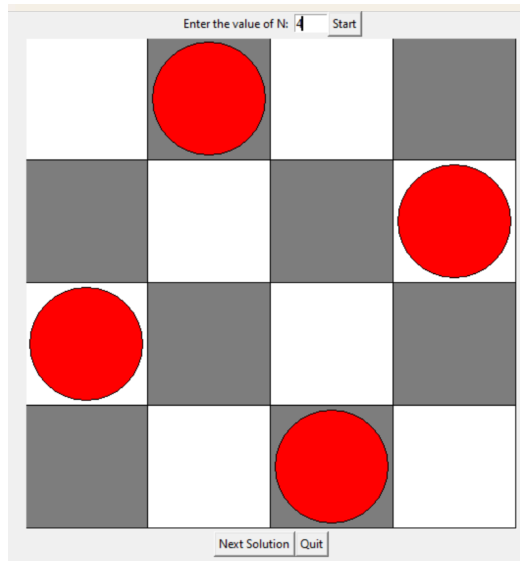
User Interface (UI)

The **Tkinter UI** interacts with the genetic algorithm and provides:

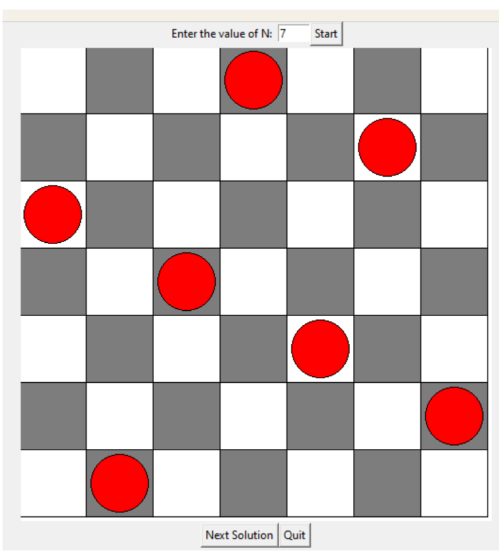
- **Input Field**: Allows the user to enter the board size (N).
- **Chessboard Visualization**: Displays solutions with queens as red circles.
- **Navigation Buttons**: Lets the user view the next solution or quit the program.

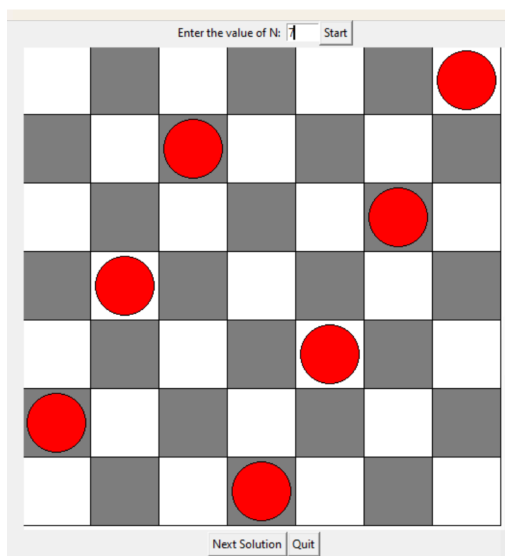
//for N=4


Enter the value of N: 4
Start
Next Solution
Quit


Enter the value of N: 4
Start
Next Solution
Quit

//for N=7


Enter the value of N: 7
Start
Next Solution
Quit


Enter the value of N: 7
Start
Next Solution
Quit

Advantages of Genetic Algorithm for N-Queens

1. **Scalability**:
    - Efficiently handles larger N compared to exhaustive search methods like backtracking.
2. **Randomized Approach**:
    - Does not rely on pre-defined patterns or rules, making it flexible.
3. **Fast Approximation**:
    - Quickly finds valid solutions even for larger problem sizes.

Limitations

1. **Non-Guaranteed Completeness**:
    - May not find all solutions due to randomness.
2. **Performance**:
    - Requires fine-tuning of parameters like `population_size` and `mutation_rate`.
3. **Computational Overhead**:
    - For very large N, the algorithm may take longer to converge.