DEPARTMENT OF COMPUTER & INFORMATION SYSTEMS ENGINEERING

Course Code: CS-323

Course Title: Artificial Intelligence

Open Ended Lab

TE Batch 2022, Fall Semester 2024

Grading Rubric

Group Members:

Student No.	Name	Roll No.
S1	AYESHA TAUFIQUE	**CS-22036**
S2	AMINA SHAHZAD	**CS-22019**
S3	ANEEBA ZAFAR	**CS-22029**

CRITERIA AND SCALES				**Marks Obtained**		
				S1	**S2**	**S3**
Criterion 1: Has the student appropriately simulated the working of the genetic algorithm?						
0	1	2	-			
The explanation is too basic.	The algorithm is explained well with an example.	The explanation is much more comprehensive.				
Criterion 2: How well is the student's understanding of the genetic algorithm?						
0	1	2	3			
The student has no understanding.	The student has a basic understanding.	The student has a good understanding.	The student has an excellent understanding.			
Criterion 3: How good is the programming implementation?						
0	1	2	3			
The project could not be implemented.	The project has been implemented partially.	The project has been implemented completely but can be improved.	The project has been implemented completely and impressively.			
Criterion 4: How good is the selected application?						
0	1	2	-			
The chosen application is too simple.	The application is fit to be chosen.	The application is different and impressive.				
Criterion 5: How well-written is the report?						
0	1	2	-			
The submitted report is unfit to be graded.	The report is partially acceptable.	The report is complete and concise.				
			Total Marks:			

GENETIC ALGORITHM FOR STRING MATCHING

PROBLEM DESCRIPTION:

- Simulate the genetic algorithm using a simple example to understand how it works.
- Produce the Python code of the genetic algorithm.
- Discover a suitable computing problem and apply the coded genetic algorithm to solve it.

ABSTRACT:

For demonstration purposes, the genetic algorithm is applied to the problem of evolving a population of strings towards a target string. The target string is predefined, and the goal of the algorithm is to evolve a population of random strings to exactly match this target.

The chosen target string is: **"Artificial Intelligence "**.

1. INRODUCTION

A genetic algorithm (GA) is a search heuristic inspired by the process of natural selection that belongs to the family of evolutionary algorithms. These algorithms are designed to find optimal or near-optimal solutions to complex optimization and search problems. Genetic algorithms mimic the process of natural evolution, employing mechanisms like selection, crossover, and mutation. These algorithms are frequently used in solving various real-world problems, including optimization of decision trees, hyper parameter tuning, machine learning, puzzle solving, and more.

This report explains the working of a genetic algorithm through a clear and detailed example using Python, along with an explanation of the theory and logic behind its implementation. We will also explain the code that executes the genetic algorithm and provide performance analysis based on real-world testing.

2. GENETIC ALGORITHM BASICS:

- **Initialization**: A population of candidate solutions (individuals) is randomly generated.
- **Selection**: The fittest individuals are selected to reproduce.
- **Population:** A group of candidate strings (individuals).
- **Fitness Function:** Measures how close a candidate string is to the target.
- **Crossover (Recombination)**: Two selected parents combine their genetic information to create offspring.
- **Mutation**: The offspring might undergo small random mutations to introduce diversity.
- **Termination**: The algorithm stops when a termination condition (such as reaching a target fitness or exceeding the maximum number of generations) is met.

3. WORKING OF THE GENETIC ALGORITHM IN PYTHON:

Below is a Python implementation of a genetic algorithm designed to solve the string matching problem. In the following sections, we will break down the code and explain its components.

3.1 Initialization of the Population

```
def initialize_population(self):
    """Create the starting population of random strings."""
    self.population = []
    for _ in range(self.population_size):
        individual = ''.join(random.choices(self.gene_pool, k=len(self.target)))
        self.population.append(individual)
```

- **Explanation**:

The `initialize_population` method generates an initial set of random solutions (strings). Each string is composed of characters randomly selected from a defined gene pool, which includes uppercase and lowercase alphabets and spaces.

- **Goal**:

We aim to evolve a population of strings toward the target string using genetic operators.

3.2 Fitness Function

```
def fitness(self, individual: str) -> int:
    """Calculate fitness based on matching characters and penalize repeated incorrect characters."""
    match_score = sum(1 for i, c in enumerate(individual) if c == self.target[i])
    diversity_penalty = len(set(individual)) / len(self.target)
    return match_score + diversity_penalty
```

Explanation: The `fitness` function calculates how close an individual solution (string) is to the target. It does so by comparing each character of the individual string with the target string:

a) **Match Score**: The number of characters that match the target string.
b) **Diversity Penalty**: A penalty is applied based on the number of unique characters in the individual to promote diversity in the population.

3.3 Parent Selection Using Tournament Selection

```
def select_parents(self):
    """Select parents using tournament selection."""
    def tournament():
        competitors = random.sample(self.population, k=5)
        return max(competitors, key=self.fitness)

    return tournament(), tournament()
```

Explanation: In the tournament selection method, two individuals are randomly chosen from the population to compete. The fittest individuals (those with higher fitness scores) are selected as parents. This process is repeated for two parents.

3.4 Crossover (Recombination)

```
def crossover(self, parent1: str, parent2: str) -> str:
    """Perform uniform crossover."""
    return ''.join(
        random.choice([gene1, gene2]) for gene1, gene2 in zip(parent1, parent2)
    )
```

Explanation: The crossover function combines two parents by selecting each gene (character) randomly from one of the two parents. This process creates offspring that inherit a combination of genetic traits from both parents.

3.5 Mutation:

```
def mutate(self, chromosome: str, stagnation: bool) -> str:
    """Mutate a chromosome with increased rates if stagnation occurs."""
    effective_rate = self.mutation_rate * (2 if stagnation else 1)
    return ''.join(
        gene if random.random() > effective_rate else random.choice(self.gene_pool)
        for gene in chromosome
    )
```

Explanation: Mutation is a small random change in the individual's genetic code. If the algorithm has been stagnating (i.e., no progress has been made for a certain number of generations), the mutation rate is increased to introduce more diversity and avoid local optima.

3.6 Elitism and Population Update

```
new_population = [self.best_solution]  # Elitism
while len(new_population) < self.population_size:
    parent1, parent2 = self.select_parents()
    child = self.mutate(self.crossover(parent1, parent2), stagnation)
    new_population.append(child)
```

Explanation: In each generation, elitism is applied where the best solution from the current generation is retained and passed on to the next generation without modification. New individuals are created by applying crossover and mutation to the selected parents.

3.7 Termination and Performance Tracking

```
if current_fitness >= len(self.target):
    print("Target reached!")
    break
```

4. EXAMPLE EXECUTION AND PERFORMANCE ANALYSIS:

The genetic algorithm starts with a randomly generated population of size 500. In each generation, the population evolves toward the target string through selection, crossover, and mutation. The process is repeated for up to 1000 generations or until the target string is found.

```
Generation 15: Best fitness = 22.608695652173914, Best solution = Artsficial Intelligence
Generation 16: Best fitness = 23.565217391304348, Best solution = Artificial Intelligence
Target reached!
Target reached!

Performance Analysis:
Total Generations: 16
Total Time: 0.67 seconds
Average Time per Generation: 0.0442 seconds
Best Solution: Artificial Intelligence, Fitness: 23.565217391304348
```

Performance Metrics:

- ✓ Total Time: 0.67 seconds
- ✓ Average Time per Generation: 0.0442 seconds
- ✓ Best Solution: "Artificial Intelligence"

5. CONCLUSION

In this report, we have demonstrated the application of a genetic algorithm to solve a simple string matching problem. The algorithm effectively evolved a population of random strings towards a target string through the use of selection, crossover, mutation, and elitism. The implementation also includes advanced features such as a diversity mechanism to prevent stagnation and dynamic mutation to improve exploration of the solution space.

This genetic algorithm could be adapted and scaled for more complex optimization problems in various domains, such as machine learning, engineering design, and operations research. The use of such algorithms can significantly improve the efficiency of solving high-dimensional, non-linear optimization problems.

FLOW OF THE PROJECT: