ARTIFICIAL INTELLIGENCE OEL

Submitted by:

Abul Hassan CS-22118

Abul Wasay CS-22126

Alman Raza CS-22127

Course Title: Artificial Inteligence

Instructor: Ms. Hameeza

Date of Submission: 28-11-2024

NED University of Engineering and Technology

GENETIC ALGORITHM FOR STRING MATCHING

What is Genetic Algorithm?

A Genetic Algorithm (GA) is an optimization technique inspired by the process of natural selection. It works by iteratively evolving a population of candidate solutions towards an optimal solution for a given problem. By mimicking biological mechanisms such as reproduction, mutation, and selection, GAs are effective in solving both constrained and unconstrained optimization problems.

Introduction

This project applies a Genetic Algorithm to the Knapsack Problem, where the objective is to select a subset of items from a list such that their total value is maximized, without exceeding a specified weight limit. Through iterative evolution, the algorithm refines its solutions to find the optimal set of items that maximizes the total value while respecting the weight constraint.

Objective

The primary goal is to evolve a population of random solutions (chromosomes) to select the best subset of items that fit into the knapsack. The algorithm evaluates and improves the population through fitness calculations, crossover, and mutation, ultimately converging on an optimal or near-optimal solution.

Genetic Algorithm for the Knapsack Problem

The genetic algorithm is implemented as follows:

1. Chromosome Representation

- Each individual solution (chromosome) represents a set of items selected or not selected for the knapsack.
- Each gene (character in the chromosome) represents an item, with values 1 (include the item) or 0 (exclude the item).
- The length of the chromosome equals the number of items.

2. Fitness Function

- The fitness function evaluates how good a solution is by calculating its total value and total weight.
- The fitness value is determined by:
    - The total value of selected items.
    - Penalizing the solutions where the total weight exceeds the knapsack's weight capacity.

- A lower fitness value indicates a solution closer to the optimal solution, with the best fitness being the highest possible value without exceeding the weight limit.

```
def cal_fitness(self):
    total_value = 0
    total_weight = 0
    for i in range(len(self.chromosome)):
        if self.chromosome[i] == 1:
            total_value += self.items[i][0]
            total_weight += self.items[i][1]

    # Penalize solutions that exceed the weight limit
    if total_weight > self.max_weight:
        return 0
    return total_value
```

3. Initial Population

- A population of 100 chromosomes (random solutions) is initialized.
- Each chromosome is randomly generated by selecting items either for inclusion (1) or exclusion (0) based on the number of items in the problem.

```
@classmethod
def mutated_genes(cls):
    # Randomly pick whether to include an item (1) or not (0)
    return random.choice([0, 1])

@classmethod
def create_gnome(cls, items):
    # Randomly generate a chromosome based on the number of items
    return [cls.mutated_genes() for _ in range(len(items))]
```

Genetic Operations

1. **Selection:**

   - The top 10% of the population (elite individuals) is passed directly to the next generation.
   - The remaining offspring are generated by mating the top 50% of the population, based on fitness.

2. **Crossover:**

   - Single-point crossover is used to combine genetic material from two parents.
   - A random crossover point is selected, and the genes from the parents are combined to form an offspring.

3. **Mutation:**

   - Each gene in the offspring has a 5% chance of being mutated (i.e., randomly changed from 0 to 1 or vice versa).
   - Mutation introduces diversity and helps prevent premature convergence to a local optimum.

```python
def mate(self, par2):
    # Perform single-point crossover
    crossover_point = random.randint(0, len(self.chromosome) - 1)
    child_chromosome = self.chromosome[:crossover_point] + par2.chromosome[crossover_point:]
    return Individual(child_chromosome, self.items, self.max_weight)
```

Termination Condition

The algorithm terminates when the best individual's fitness reaches zero (indicating a perfect solution) or after a set number of generations.

Implementation

The algorithm was implemented in Python. Below is a summary of the key components:

Class Definition

- The Individual class represents a single solution (chromosome).
- It includes methods for creating chromosomes, calculating fitness, and performing genetic operations (crossover and mutation).

Driver Code

- The main program initializes the population and iteratively applies genetic operations (selection, crossover, mutation) until a solution is found or a stopping condition is met.

```
while not found:
    # Sort the population by fitness (descending order)
    population = sorted(population, key=lambda x: x.fitness, reverse=True)

    # Check if the best individual has been found
    if population[0].fitness > 0:
        found = True
        break

    # Create a new generation using elitism and crossover
    new_generation = []

    # Perform elitism: top 10% of the population goes to the next generation
    s = int((10 * population_size) / 100)
    new_generation.extend(population[:s])

    # Perform crossover for the remaining individuals
    s = int((90 * population_size) / 100)
    for _ in range(s):
        parent1 = tournament_selection(population)
        parent2 = tournament_selection(population)
        child = parent1.mate(parent2)
        new_generation.append(child)

    population = new_generation
```

Intermediate results

(e.g., best fitness in each generation) are displayed to track the algorithm's progress.

Output

```
Enter the maximum weight capacity of the knapsack: 50
Enter the number of items: 5
Enter value for item 1: 100
Enter weight for item 1: 10
Enter value for item 2: 80
Enter weight for item 2: 20
Enter value for item 3: 90
Enter weight for item 3: 10
Enter value for item 4: 50
Enter weight for item 4: 10
Enter value for item 5: 100
Enter weight for item 5: 5
Enter the population size: 100

Final Result:
Best Knapsack Value: 370
Selected Items: [1, 2, 3, 5]
Total Value: 370
Total Weight: 45
```

Performance Analysis

Strengths:

- The algorithm effectively converges toward an optimal or near-optimal solution for the knapsack problem.
- Mutation ensures diversity, helping to avoid local optima.
- The crossover operation efficiently combines the best features of parents to generate offspring with better potential.

Limitations:

- The algorithm's performance can vary based on the randomness of the initial population and the selection of parameters like mutation rate and population size.
- For larger numbers of items, the algorithm may take longer to converge due to the increased search space.

Conclusion

This project demonstrates the effectiveness of Genetic Algorithms in solving the Knapsack Problem. By evolving a population of candidate solutions through genetic operations such as selection, crossover, and mutation, the algorithm successfully identifies a near-optimal solution for maximizing the knapsack's value while adhering to the weight constraint

Future Enhancements

- **Dynamic Mutation Rates:**

  Adjust mutation probabilities based on generation or fitness trends to improve convergence speed.

- **Parallel Processing:**

  Implement multi-threading to speed up the evolution process, particularly for larger problem sizes.

- **Advanced Selection Methods:**

  Explore alternative selection methods, such as tournament selection, to improve the quality of parents selected for reproduction.

- **Hybrid Approaches:**

  Combine Genetic Algorithms with other optimization techniques (e.g., simulated annealing or local search) to enhance solution quality and convergence speed.