Multi-Verse Optimization for the Traveling Salesman Problem

Helia Ghorbani
Ghazal Pouresfaniyar

Overview of the Algorithm:

The Multi-Verse Optimization (MVO) algorithm is a metaheuristic algorithm inspired by the concept of a multiverse, which is a hypothetical set of multiple possible universes. The algorithm is used to solve optimization problems, such as the Traveling Salesman Problem (TSP).

The MVO algorithm works as follows:

1. **Initialization**: The algorithm initializes a population of random solutions, called universes. In the case of the TSP, each universe is a permutation of the indices of the cities, representing a tour through the cities.

2. **Fitness evaluation:** The algorithm evaluates the fitness of each universe by calculating the total distance traveled by the tour represented by that universe.

3. **Main loop:** The algorithm performs a certain number of iterations, during which it updates the universes by swapping two cities in them. The probability of selecting a universe for the update is proportional to its fitness, with a higher fitness leading to a higher probability of selection. At each iteration, the wormhole existence probability is updated.

4. **Wormhole existence probability update:** The wormhole existence probability determines the probability of a universe being selected for the update. It is updated at each iteration of the algorithm using the formula w_ep = w_ep * (1 - t / num_iterations), where t is the current iteration and num_iterations is the total number of iterations.

5. **Universe update:** For each universe, the algorithm selects a random universe from the population with a probability proportional to its fitness. It then creates a new universe by swapping two cities in the selected universe, and calculates the fitness

of the new universe. If the fitness of the new universe is better than the fitness of the original universe, the original universe is replaced with the new universe.

6. **Termination**: After the specified number of iterations, the algorithm terminates and returns the best solution found, which is the universe with the lowest fitness value.

The MVO algorithm is a population-based optimization algorithm that uses a probabilistic selection mechanism to update the solutions. It is a metaheuristic algorithm, which means that it does not guarantee to find the optimal solution, but it is capable of finding good solutions for a wide range of optimization problems. The MVO algorithm has been successfully applied to various optimization problems, including the TSP.

Code Explanation:

```
import numpy as np
import random
import matplotlib.pyplot as plt
```

This code block imports the necessary libraries for the implementation of the algorithm. NumPy is used for mathematical operations, random for generating random numbers, math for mathematical functions, and matplotlib for visualization.

```
# Euclidean distance function
def euclidean_distance(point1, point2):
    return np.sqrt(np.sum((np.array(point1) - np.array(point2)) ** 2))
```

This function calculates the Euclidean distance between two points in two-dimensional space using the formula sqrt((x1-x2)^2 + (y1-y2)^2). It takes as input two points represented as tuples or lists.

```
# Initialize a random TSP problem
def create_tsp_problem(num_cities):
    # cities = [np.random.rand(2) for _ in range(num_cities)]
    # integer(less than 10) input
    cities = [np.random.randint(10, size=2) for _ in range(num_cities)]
```

```
    return cities
```

This function generates a random TSP problem consisting of a given number of cities. It generates a list of 2D coordinates for the cities, with each coordinate being an integer between 0 and 9.

```
# Calculate the total distance of a TSP tour
def total_distance(cities, tour):
    return sum(euclidean_distance(cities[tour[i]], cities[tour[i - 1]]) for
i in range(len(tour)))
```

This function calculates the total distance of a given tour through the cities, where a tour is a permutation of the indices of the cities. It does so by summing the Euclidean distances between consecutive cities in the tour. The cities argument is a list of 2D coordinates, and the tour argument is a list of integers representing the indices of the cities in the tour.

```
# Generate a random tour for the TSP problem
def random_tour(num_cities):
    tour = list(range(num_cities))
    random.shuffle(tour)
    return tour
```

This function generates a random tour through a given number of cities. It generates a list of integers representing the indices of the cities in the tour and shuffles them randomly.

```
# MVO algorithm function
def mvo_tsp(cities, population_size, num_iterations, w_ep_initial=1):
    # Applies the MVO algorithm to solve the TSP problem
    num_cities = len(cities)
    # Initialize a population of random tours
    universes = [random_tour(num_cities) for _ in range(population_size)]
    # Calculate the fitness of each tour
    fitness = [total_distance(cities, universe) for universe in universes]
    # Initialize the exploration probability
    w_ep = w_ep_initial
    # Run the MVO algorithm for a fixed number of iterations
```

```
    for t in range(num_iterations):
        # Find the best and worst fitness values in the population
        best_fitness = min(fitness)
        worst_fitness = max(fitness)
        # Normalize the fitness values
        if worst_fitness == best_fitness:
            norm_fitness = [1.0] * len(fitness)
        else:
            norm_fitness = [(worst_fitness - fit) / (worst_fitness -
best_fitness) for fit in fitness]
        # Calculate the probability of each universe to act as a wormhole
        wormhole_probs = [norm_fit / sum(norm_fitness) for norm_fit in
norm_fitness]
        # Calculate the cumulative probability of each universe to act as a
wormhole
        total_wormhole_probs = [sum(wormhole_probs[0:i + 1]) for i in
range(population_size)]
        # Perform the wormhole operation for each universe
        for i in range(population_size):
            # Randomly choose to perform the wormhole operation or not
            rand_prob = random.random()
            if rand_prob < w_ep:
                # Select a universe to perform the wormhole operation
                try:
                    selected_universe_index = next(i for i, prob in
enumerate(total_wormhole_probs) if prob >= rand_prob)
                except StopIteration:
                    selected_universe_index = np.random.randint(0,
len(total_wormhole_probs))
            else:
                selected_universe_index = i
            # Perform the wormhole operation to create a new universe
            new_universe = universes[selected_universe_index].copy()
            idx1, idx2 = random.sample(range(num_cities), 2)
            new_universe[idx1], new_universe[idx2] = new_universe[idx2],
new_universe[idx1]
            # Calculate the fitness of the new universe
            new_fitness = total_distance(cities, new_universe)
            # Update the current universe if the new universe has better
fitness
            if new_fitness < fitness[i]:
                universes[i] = new_universe
                fitness[i] = new_fitness
```

```
        # Update the exploration probability
        w_ep = w_ep * (1 - t / num_iterations)
    # Find the best solution in the population
    best_solution_index = fitness.index(min(fitness))
    best_solution = universes[best_solution_index]
    best_solution_fitness = fitness[best_solution_index]
    # Return the best solution and its fitness value
    return best_solution, best_solution_fitness
```

This function implements the MVO (Multiverse Optimizer) algorithm to solve the TSP. The main steps of the algorithm are:

1. Initialize a population of random tour solutions. Each universe represents a tour.
2. Calculate the fitness (total distance) of each tour.
3. Initialize the exploration probability w_ep. This controls the probability of performing the wormhole operation.
4. For a fixed number of iterations:

- Find the best and worst fitness in the population.
- Normalize the fitness values. If the best and worst fitnesses are equal, set all normalized fitnesses to 1.
- Calculate the probability of each universe acting as a wormhole based on its normalized fitness.
- Calculate the cumulative probability of each universe acting as a wormhole.
- For each universe:
    - Randomly choose to perform the wormhole operation with probability w_ep.
    - If doing so, select a wormhole universe based on the cumulative probabilities.
    - Perform the wormhole operation on the selected universe, by swapping two cities in the tour.
    - Calculate the fitness of the new tour.
    - If the new fitness is better, replace the current universe with the new one.
- Update the exploration probability w_ep.

5. Find the best solution (tour with minimum fitness) in the population.
6. Return the best solution and its fitness.

So in summary, the algorithm maintains a population of tour solutions and performs the wormhole operation to explore new solutions, keeping the best ones. This allows it to converge to an optimal or near-optimal TSP tour.

```
# Plot the TSP tour
def plot_tour(cities, best_tour):
    x = [city[0] for city in cities]
    y = [city[1] for city in cities]

    fig, ax = plt.subplots()
    ax.plot(x, y, 'ro')
    for i, txt in enumerate(range(len(cities))):
        ax.annotate(txt, (x[i], y[i]), textcoords="offset points",
xytext=(0, 10), ha='center')
    for i in range(len(best_tour)):
        ax.plot([x[best_tour[i-1]], x[best_tour[i]]], [y[best_tour[i-1]],
y[best_tour[i]]], 'b')
    plt.show()
```

This function visualizes the best tour found by the algorithm. It takes as input the cities list of 2D coordinates and the best_tour list of integers representing the indices of the cities in the best tour. It plots the cities as red dots and the tour as a blue line, and annotates each city with its index. The resulting plot is displayed using the matplotlib library.

```
# This plot displays the distances between the cities in the TSP problem.
# It can be used to verify that the distance calculations are correct.
# The plot is particularly useful when the number of cities is relatively
small,
# as it allows for a clearer visualization of the distances.
def plot_tour2(cities, best_tour):
    # Plots a TSP tour using Matplotlib
    plt.scatter([city[0] for city in cities], [city[1] for city in cities],
color='red', zorder=1)
    for i in range(len(best_tour)):
        # Get the coordinates of the current city and its predecessor
        current_city = cities[best_tour[i]]
        prev_city = cities[best_tour[i-1]]
        # Calculate the distance between the current city and its
predecessor
        distance = euclidean_distance(current_city, prev_city)
        # Calculate the midpoint between the current city and its
predecessor
        midpoint = ((current_city[0] + prev_city[0]) / 2, (current_city[1]
```

```
+ prev_city[1]) / 2)
        # Add a line between the current city and its predecessor
        plt.plot([current_city[0], prev_city[0]], [current_city[1],
prev_city[1]], 'k-', zorder=0)
        # Add an annotation with the distance at the midpoint between the
current city and its predecessor
        plt.annotate(str(round(distance, 2)), xy=midpoint, xytext=(5, 5),
textcoords='offset points')
        # Add an annotation with the city index at the current city
        plt.annotate("v"+str(best_tour[i]), xy=current_city, xytext=(5, 5),
textcoords='offset points')
    plt.show()
```

This function plots the TSP tour in a way that clearly shows the distances between cities. This makes it easy to visually verify that the distance calculations are correct.

The main steps of the function are:

1. It plots all the cities as red scatter points.
2. For each city in the tour:

- It gets the coordinates of the current city and its predecessor city.
- It calculates the Euclidean distance between the current city and its predecessor using the euclidean_distance() function.
- It calculates the midpoint between the current city and its predecessor.
- It plots a line between the current city and its predecessor.
- It adds an annotation at the midpoint showing the calculated distance, rounded to 2 decimal places.
- It adds an annotation at the current city showing its index (v0, v1, etc).

3. It displays the plot using plt.show().

So in summary, this function plots:

- The cities as points
- Lines between consecutive cities in the tour
- Annotations showing the distances between consecutive cities
- Annotations showing the index of each city

This makes it easy to visually verify that:

- The lines are plotted between the correct cities
- The calculated distances match the distances seen on the plot

This helps debug and verify the distance calculation and TSP solution. The visualization is particularly useful when the number of cities is small, as in the example usage in the code.

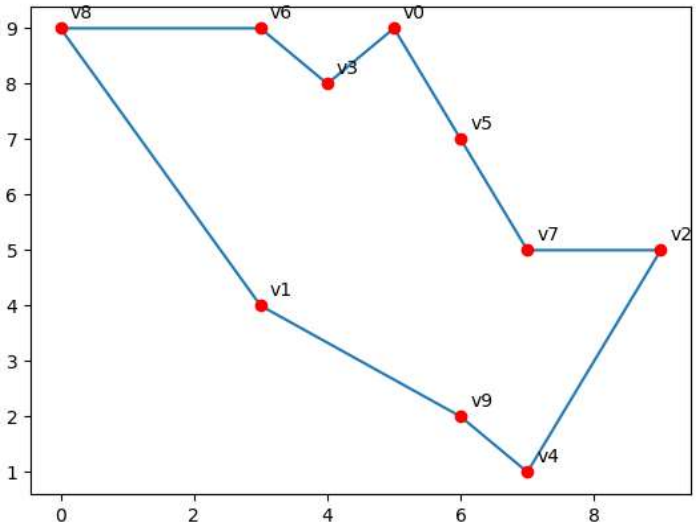
Sample Output:

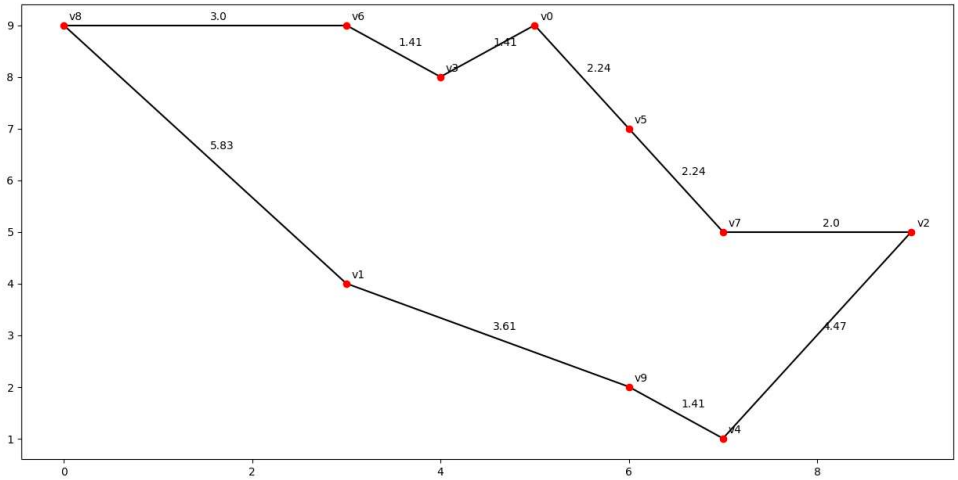
Here's an example of what the output may look like:

```
Best tour: [4, 2, 7, 5, 0, 3, 6, 8, 1, 9]
Best distance: 27.623415767427737
```

Note that the output may vary each time you run the code, as the algorithm starts with randomly generated tours and updates the universes using random probabilities.

The plot generated by the code shows the best tour found by the MVO algorithm for the TSP problem. The cities are represented by red dots, and the lines connecting the cities represent the order in which the salesman visits them.





GitHub Link:

https://github.com/haxhex/MVO-TSP

Also, you can see more information about the project details like implemented algorithm and integration part in this repository.