Department of Computer Science, Faculty of Sciences and Technologies, Tangier

FLOW REGIME ALGORITHM

Implemetation on traveling salseman problem

Presented by:

Aachabi Mohammed

1

INTRODUCTION

Overview of TSP

INTRODUCTION

Overview of TSP

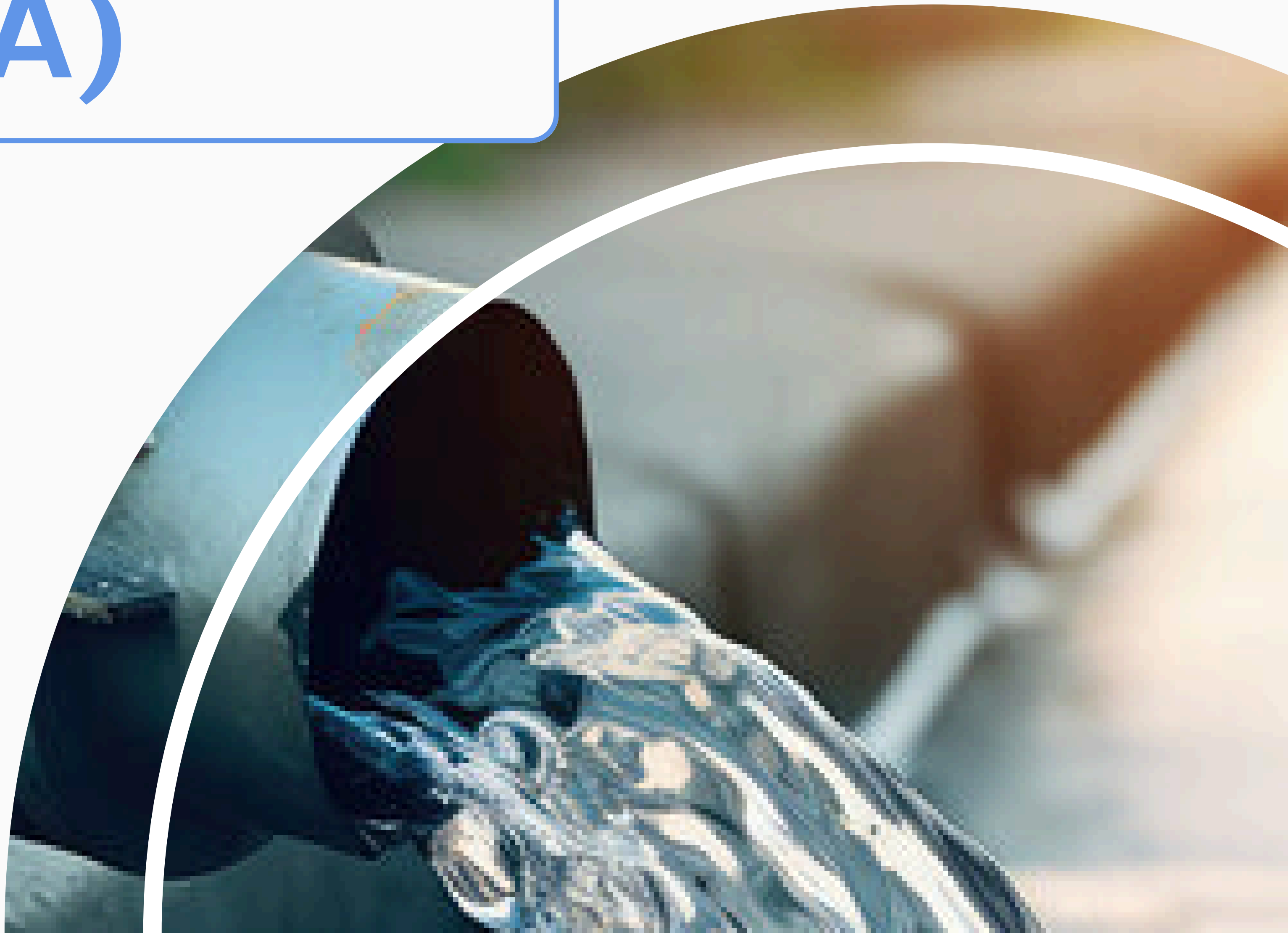
The Traveling Salesman Problem (TSP) stands as one of the most enduring and well-known combinatorial optimization challenges in the realms of computer science, operations research, and mathematics.

The TSP involves determining the shortest possible route that visits a set of cities exactly once and returns to the origin city.

The problem is how to find a minimal route passing from all the nodes. for example, if you take path one from {A, B, C, D, E,A} and the path two that is {A, B, C, E, D, A} you have passed all the cities, but the two paths are diffrent in the name of the distance .

TSP focus on reducing cost of building construction engineering and also reduces material wastages, through its principals of finding the minimum cost path of the salesman.

2 FLOW REGIM Algorithm (FRA)

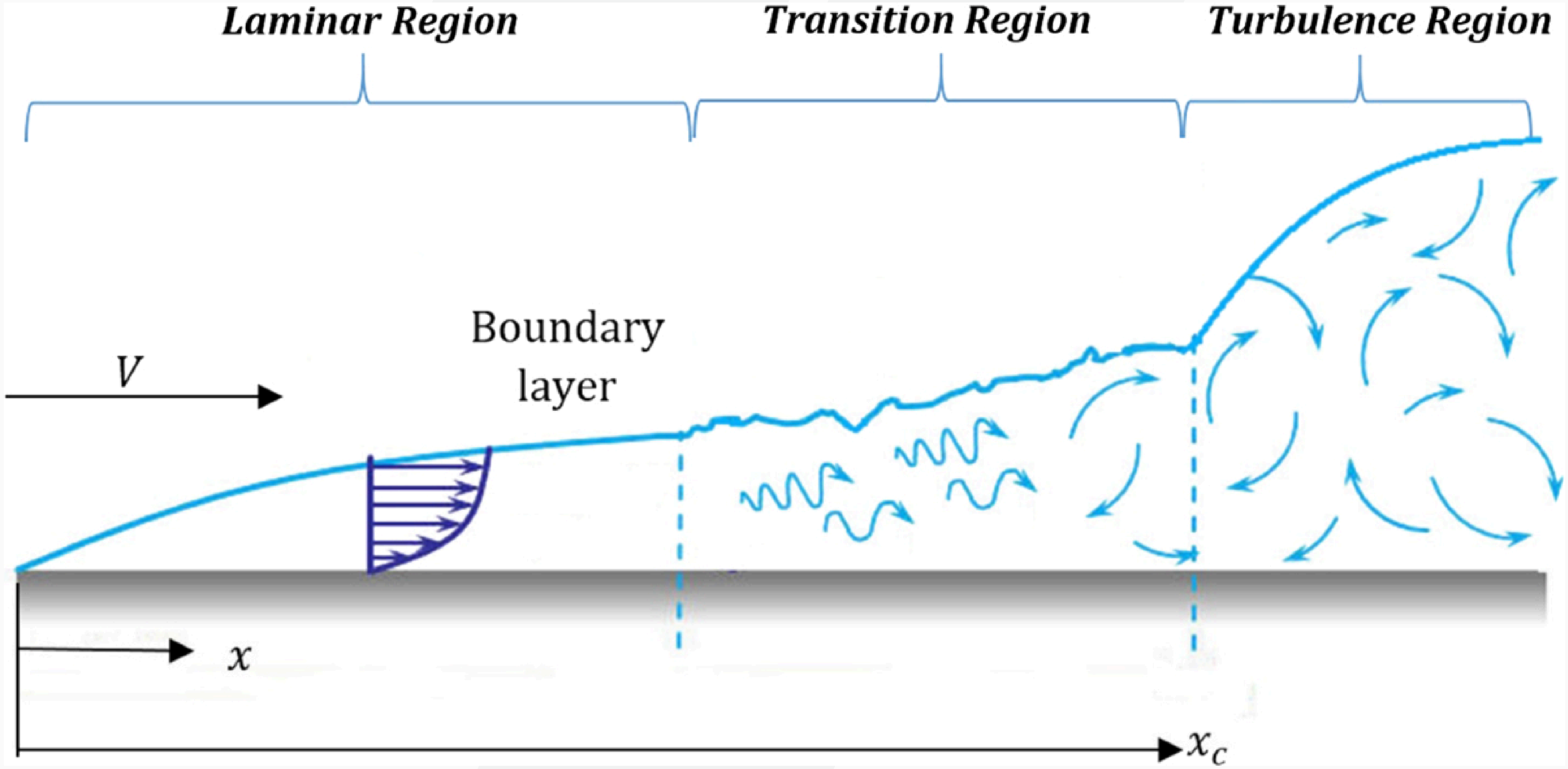


FLOW REGIME ALGORITHM

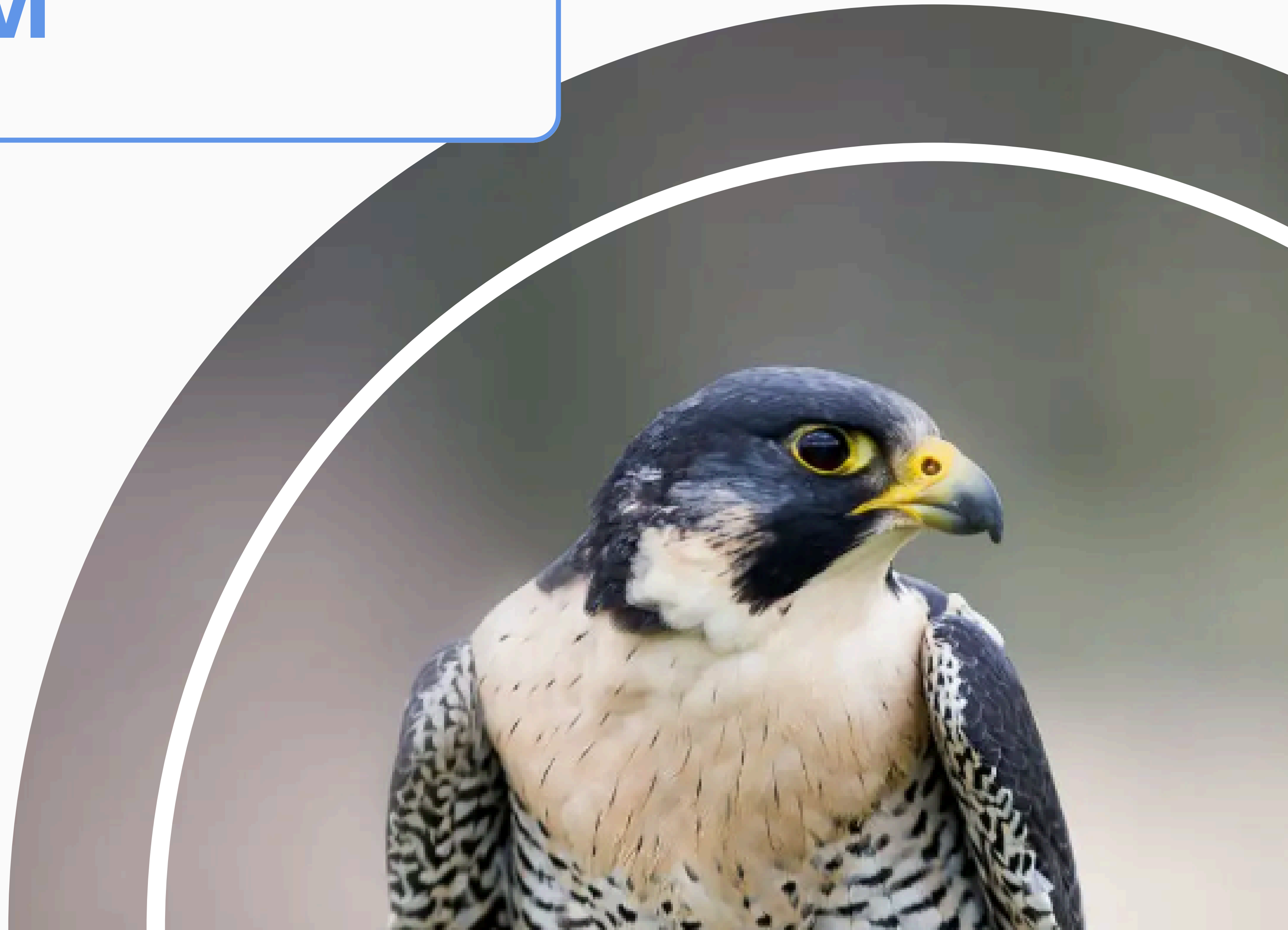
Overview of FRA

Flow Regime Algorithm (FRA), a physics-based optimization algorithm,.

The main sources of inspiration are classical fluid mechanics and flow regimes. The flow regime usually is being divided into two categories which are laminar and turbulent flows. Reynolds number is the parameter which defines that the flow regime is laminar or turbulent.



2 TSP BASED ON FLOW REGIM ALGORITHM



STEPS OF FLOW REGIM ALGORITHM ON TSP

- **Generate Initial Population**

Set the parameters for FOA: number of falcons, alpha, beta, delta, p_a, p_d, and maximum iterations.

$$\text{population}[i] = \text{random.sample}(\{0, 1, \ldots, \text{num\_cities} - 1\}, \text{num\_cities})$$

Inputs:

- num_individuals (int): Number of candidate solutions (routes) to generate.
- num_cities (int): Number of cities in the TSP instance.

Outputs:

- population (list of lists): A list where each element is a permutation of city indices representing a candidate route.

```python
def generate_initial_population(num_individuals, num_cities):
    population = []
    for _ in range(num_individuals):
        permutation = random.sample(range(num_cities), num_cities)
        population.append(permutation)
    return population
```

STEPS OF FALCON OPTIMIZATION ALGORITHM

- **Calculate Fitness**

$$\text{fitness} = \sum_{k=0}^{n-1} d(\text{permutation}[k], \text{permutation}[(k+1)\%n])$$

$$d(i,j) = \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2}$$

Inputs:

- permutation (list of ints): A permutation of city indices representing a route.
- city_locations (list of tuples): Coordinates of the cities.

Outputs:

- fitness (float): The total distance of the route.

```
def calculate_fitness(permutation, city_locations):
    total_distance = 0
    for i in range(len(permutation) - 1):
        city1 = city_locations[permutation[i]]
        city2 = city_locations[permutation[i + 1]]
        total_distance += distance(city1, city2)
    city1 = city_locations[permutation[-1]]
    city2 = city_locations[permutation[0]]
    total_distance += distance(city1, city2)
    return total_distance
```

STEPS OF FALCON OPTIMIZATION ALGORITHM

- **Calculate Reynolds Number**

  Calculate the Reynolds number to determine the flow regime

$$Re = \frac{\rho \cdot v \cdot L}{\mu}$$

ρ = density

v = velocity

L = characteristic length

μ = viscosity

Inputs:

- velocity (float): The velocity of the flow.

Outputs:

- Re (float): The Reynolds number.

```
def reynolds_number(velocity):
    return (DENSITY * velocity * CHARACTERISTIC_LENGTH) / VISCOSITY
```

STEPS OF FALCON OPTIMIZATION ALGORITHM

- **Generate Levy and Gaussian Distributions**

$$\text{Levy distribution: } \mathrm{Levy} \sim \text{standard cauchy distribution}$$

$$\text{Gaussian distribution: } \mathrm{Gaussian} \sim \mathcal{N}(0,1)$$

Inputs: None (random distributions).

Outputs:

- Levy Distribution: A random number from the standard Cauchy distribution.
- Gaussian Distribution: A random number from the normal distribution with mean 0 and standard deviation 1.

```python
def levy_distribution():
    return np.random.standard_cauchy()

def gaussian_distribution():
    return np.random.normal()
```

STEPS OF FALCON OPTIMIZATION ALGORITHM

- **Update Positions**

Inputs:
- population, city_locations, alpha, beta, gamma

Outputs
- new_population.

For laminar flow ($Re < 2000$):

$$\text{new\_position}[i] = (\text{position}[i] + \text{int}(\text{Levy step}))\%\text{num\_cities}$$

For turbulent flow ($Re \geq 2000$):

$$\text{new\_position}[i] = (\text{position}[i] + \text{int}(\text{Gaussian step}))\%\text{num\_cities}$$

Flow regime attraction (probability α):

$$\text{if random.random()} < \text{alpha: new\_position[i]} = \text{best\_neighbor\_index}$$

Random disturbance (probability β):

$$\text{if random.random()} < \text{beta: swap(new\_position[i], new\_position[randon}$$

```python
def update_positions(population, city_locations, alpha, beta, gamma):
    new_population = []
    for individual in population:
        new_position = individual[:]
        velocity = gamma * gaussian_distribution()
        re_number = reynolds_number(velocity)
        for i in range(len(individual)):
            if re_number < 2000:  # Laminar flow
                levy_step = levy_distribution()
                new_position[i] = (individual[i] + int(levy_step)) % len(city_locations)
            else:  # Turbulent flow
                random_step = gaussian_distribution()
                new_position[i] = (individual[i] + int(random_step)) % len(city_locations)
            if random.random() < alpha:
                best_neighbor_index = find_best_neighbor(individual, city_locations)
                new_position[i] = individual[best_neighbor_index]
            elif random.random() < beta:
                random_index = random.randint(0, len(individual) - 1)
                new_position[i], new_position[random_index] = new_position[random_index],
        new_population.append(new_position)
    return new_population
```

STEPS OF FALCON OPTIMIZATION ALGORITHM

- **Select Best Individuals**

Select the top individuals based on fitness.

$$\text{sorted\_individuals} = \text{sorted}(\text{zip}(population, fitness_values), \text{key} = \lambda x : x[1]$$

$$\text{selected\_individuals} = [\text{individual for individual, \_ in sorted\_individuals}[:\text{nu}$$

Inputs:

- population (list of lists): Current population of candidate solutions.
- fitness_values (list of floats): Fitness values of the population.
- num_individuals_to_select (int): Number of top individuals to select.

Outputs:

- selected_individuals (list of lists): Top num_individuals_to_select individuals based on fitness.

```python
def select_best_individuals(population, fitness_values, num_individuals_to_select):
    """Selects the top 'num_individuals_to_select' individuals based on fitness."""
    sorted_individuals = sorted(zip(population, fitness_values), key=lambda x: x[1])
    return [individual for individual, _ in sorted_individuals[:num_individuals_to_select]]
```

MAIN LOOP

- **Calculate Fitness for Each Individual:**

```
# Calculate fitness for each individual
fitness_values = [calculate_fitness(permutation, city_locations) for permutation in population]
```

- **Select the Best Individuals**

```
# Select the best individuals
population = select_best_individuals(population, fitness_values, num_individuals)
```

- **Update Positions**

```
# Update positions
population = update_positions(population, city_locations, alpha, beta, gamma)
```

- **Recalculate Fitness for Updated Population:**

```
# Calculate fitness for the updated population
fitness_values = [calculate_fitness(permutation, city_locations) for permutation in population]
```

- **Update Best Solution**

```
# Update best solution
for i in range(num_individuals):
    fitness = fitness_values[i]
    if fitness < best_fitness:
        best_solution = population[i]
        best_fitness = fitness
```

RESULTS

On the file **berlin54.tsp**

```
Final Best Solution: [22, 40, 9, 44, 28, 41, 20, 1, 15, 5, 18, 32, 37, 31, 0, 43, 6, 2, 16, 7, 47, 34, 42, 3, 29, 48, 33, 35, 36, 49, 8, 10, 19,
25, 50, 12, 13, 4, 46, 11, 51, 21, 17, 38, 27, 45, 24, 14, 23, 39, 26, 30]
Final Best Distance: 23874.59299811015
```

On the file **a29.tsp**

```
Final Best Solution: [20, 28, 18, 23, 26, 1, 0, 2, 4, 15, 12, 3, 6, 5, 13, 11, 7, 14, 8, 10, 16, 19, 17, 24, 27, 22, 25, 21, 9]
Final Best Distance: 1137.8453511675864
```

Interpretation

- **Convergence:** The fitness values show improvement, indicating the algorithm's success in finding shorter routes over iterations.
- **Optimization:** The final best distance indicates the quality of the solution found. In the context of TSP, this distance represents the length of the shortest route discovered by the FRA.

CONCLUSION

The Flow Regime Algorithm (FRA) applied to the Traveling Salesman Problem (TSP) shows promising results. It was able to identify a route with a total distance of 23874.59 units in 90 generations. This demonstrates the potential of FRA to provide competitive solutions to the TSP by leveraging principles inspired by fluid dynamics and flow regimes.

LIMITATIONS

- **Local Optima:** Like many metaheuristic algorithms, FRA may get trapped in local optima, especially if the search space is large and complex.
- **Parameter Sensitivity:** The performance of FRA is sensitive to its parameters (e.g., alpha, beta, gamma). Fine-tuning these parameters can be challenging and may require extensive experimentation.
- **Scalability:** The computational cost and time required by FRA increase with the number of cities. For significantly larger TSP instances, the algorithm might require more iterations and computational resources.
- **Diversity Maintenance:** Ensuring diversity in the population to avoid premature convergence can be difficult. Strategies to maintain diversity are crucial for the algorithm's success.
- **Reynolds Number Threshold:** The choice of 2000 as the threshold for distinguishing between laminar and turbulent flow is based on fluid dynamics principles. However, its direct application to TSP might need further justification and adaptation.
- **Randomness and Repeatability:** Due to the inherent randomness in the algorithm (e.g., Levy and Gaussian distributions), different runs may yield different results. This can be mitigated by running the algorithm multiple times and averaging the results.

Merci pour votre attention