```
import numpy as np
import random
import math

# Objective function: calculate the total distance of a tour (path)
def calculate_distance(tour, distance_matrix):
    distance = 0
    for i in range(len(tour) - 1):
        distance += distance_matrix[tour[i], tour[i + 1]]
    distance += distance_matrix[tour[-1], tour[0]]  # Return to the start
    return distance

# Generate the initial population (nests) of city permutations
def initialize_population(num_nests, num_cities):
    population = []
    for _ in range(num_nests):
        nest = np.random.permutation(num_cities)
        population.append(nest)
    return np.array(population)

# Update a nest using Levy flight
def levy_flight(nest, alpha=1.5):
    # Swap two random cities in the permutation (simplified levy flight for TSP)
    i, j = random.sample(range(len(nest)), 2)
    new_nest = nest.copy()
    new_nest[i], new_nest[j] = new_nest[j], new_nest[i]
    return new_nest

# Main Cuckoo Search algorithm for TSP
def cuckoo_search(cities, num_nests=20, max_iter=100, p_a=0.1, alpha=1.5):
    num_cities = len(cities)
    distance_matrix = np.linalg.norm(cities[:, np.newaxis] - cities, axis=2)  # Calculate distance matrix
    nests = initialize_population(num_nests, num_cities)
    fitness = np.array([1 / (1 + calculate_distance(nest, distance_matrix)) for nest in nests])

    best_nest = nests[np.argmin(fitness)]
    best_fitness = np.min(fitness)

    for iteration in range(max_iter):
        new_nests = nests.copy()
        for i in range(num_nests):
            new_nests[i] = levy_flight(nests[i], alpha)
            new_fitness = 1 / (1 + calculate_distance(new_nests[i], distance_matrix))

            # If new solution is better, update the nest
            if new_fitness > fitness[i]:
                nests[i] = new_nests[i]
                fitness[i] = new_fitness

        # Perform random replacement of some nests
        if random.random() < p_a:
            random_idx = np.random.randint(num_nests)
            nests[random_idx] = np.random.permutation(num_cities)
            fitness[random_idx] = 1 / (1 + calculate_distance(nests[random_idx], distance_matrix))

        # Update the best solution
        current_best_idx = np.argmin(fitness)
        current_best_fitness = fitness[current_best_idx]

        if current_best_fitness < best_fitness:
            best_fitness = current_best_fitness
            best_nest = nests[current_best_idx]

    return best_nest, 1 / best_fitness  # Return best solution and its corresponding distance

# Plot the best solution
def plot_solution(cities, best_tour):
    best_cities = cities[best_tour]
    plt.plot(best_cities[:, 0], best_cities[:, 1], 'bo-', markersize=6)
    plt.scatter(cities[:, 0], cities[:, 1], color='red', marker='x')
    for i, city in enumerate(cities):
        plt.text(city[0], city[1], f'{i}', fontsize=12, ha='right')
    plt.title("Cuckoo Search TSP Solution")
    plt.show()

# Example Usage
```

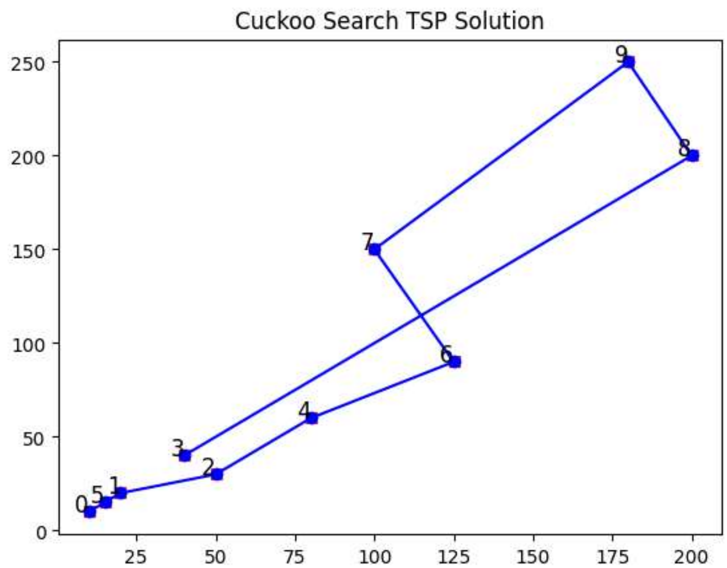
```
if __name__ == "__main__":
    cities = np.array([
        [10, 10], [20, 20], [50, 30], [40, 40], [80, 60],
        [15, 15], [125, 90], [100, 150], [200, 200], [180, 250]
    ])

    best_tour, best_length = cuckoo_search(cities, num_nests=20, max_iter=100, p_a=0.1)
    print(f"Best tour order: {best_tour}")
    print(f"Best tour length: {best_length:.2f}")

    plot_solution(cities, best_tour)

print("Vatsal - 1BM22CS323")
```

```
Best tour order: [0 5 1 2 4 6 7 9 8 3]
Best tour length: 1644.99
```




```
Vatsal - 1BM22CS323
```

Start coding or generate with AI.