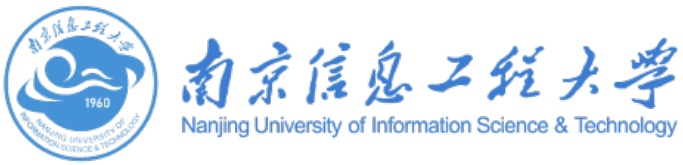


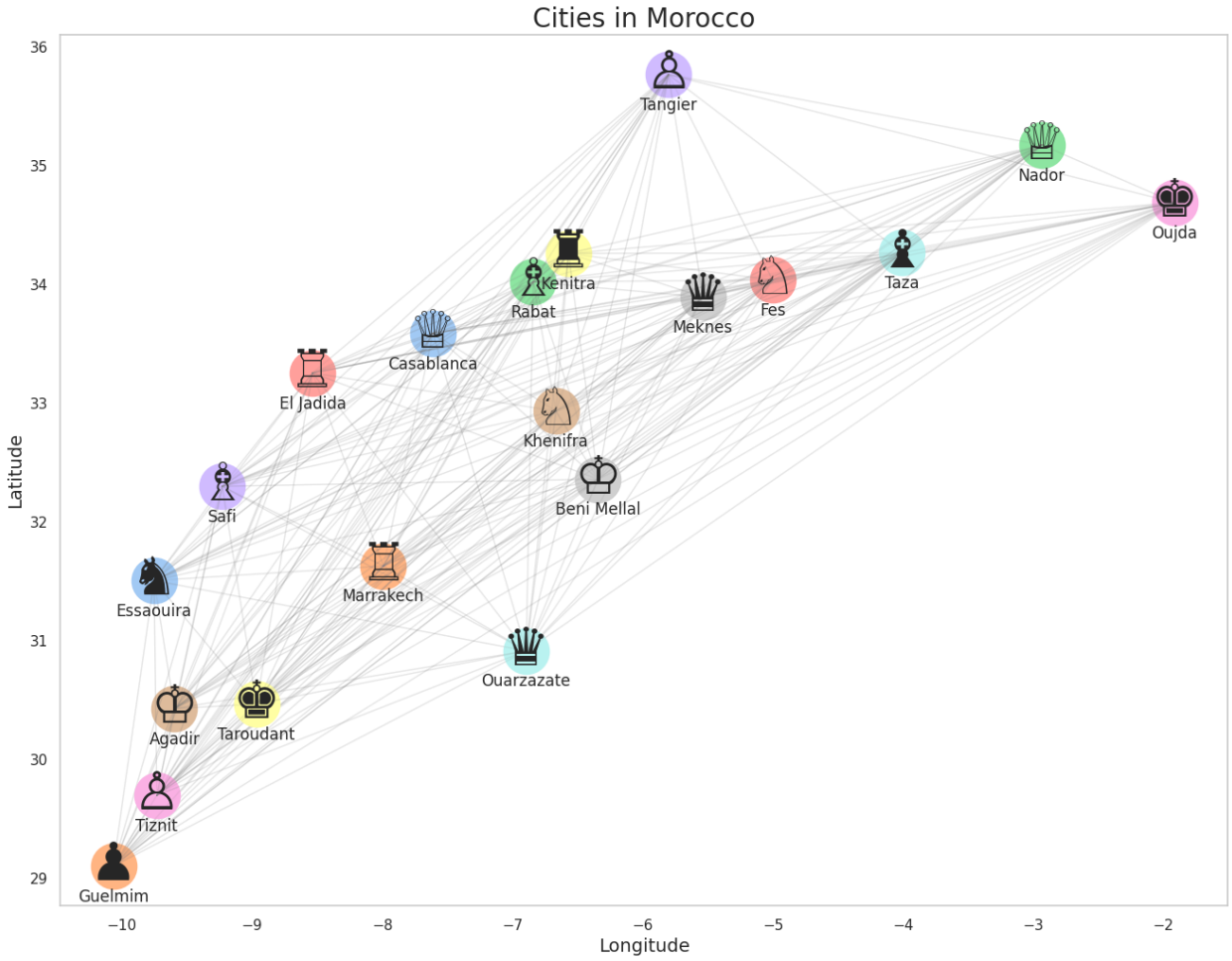
EXP-01:
Traveling Salesman Problem (TSP) using Genetic Algorithm (GA)

Abstract

This study explores the use of a Genetic Algorithm (GA) to solve the Traveling Salesman Problem (TSP) for 20 cities in Morocco, including major urban centers like Casablanca, Marrakech, and Rabat. The goal was to find the shortest route that visits each city once and returns to the starting point, a known NP-hard optimization problem. The GA was implemented in Python, using libraries like NumPy for numerical computations and Matplotlib with Seaborn for visualizations. The algorithm employed a population size of 250, a crossover rate of 0.8, and a mutation rate of 0.2, evolving over 200 generations. Key features included a fitness function based on Euclidean distances, a custom crossover operator to ensure valid TSP paths, a mutation operator to maintain genetic diversity, and an elitism strategy preserving the top 10% of solutions. City coordinates were derived from longitude and latitude to ensure accurate distance measurements, enabling consistent route optimization through generations. The algorithm's performance was tracked by reporting the best fitness value every ten generations, eventually converging on an optimal or near-optimal solution, visually displayed on a 2D map. Although the results were promising, further improvements, such as fine-tuning algorithm parameters and experimenting with alternative operators or local search methods, were suggested. This experiment demonstrates the effectiveness of evolutionary algorithms in solving complex optimization problems, combining insights from computer science, operations research, and geography, with broader applications in similar routing challenges.

Introduction

This report details a class experiment aimed at solving the Traveling Salesman Problem (TSP) using a Genetic Algorithm (GA). The TSP is a well-known optimization problem in computer science and operations research, where the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city.



Problem Description

In this experiment, we focused on 20 cities in Morocco. The objective was to find the optimal path that visits all these cities while minimizing the total distance traveled.

Dataset

The dataset consists of 20 Moroccan cities, each represented by its name and geographical coordinates (longitude and latitude). Here's a snippet of how the data is structured in my code:

```
cities_names = ["Casablanca", "Marrakech", "Rabat", "Fes", "Tangier", "Agadir", "Oujda",
"Meknes", "Kenitra", "Taza","Essaouira", "Guelmim", "Nador", "El Jadida", "Safi",
"Khenifra", "Tiznit", "Beni Mellal", "Taroudant", "Ouarzazate"]

x = [-7.61138, -7.99205, -6.84165, -5.00229, -5.80214, -9.59811, -1.91282, -5.53763,
-6.57001, -4.00767, -9.75435, -10.06667, -2.93330, -8.53700, -9.23000, -6.66667, -9.73320,
-6.34300, -8.96700, -6.89500]

y = [33.58831, 31.62867, 34.02088, 34.03724, 35.76785, 30.42776, 34.68940, 33.89379,
34.26167, 34.26578, 31.50633, 29.10000, 35.17450, 33.25400, 32.30000, 32.93333, 29.69690,
32.36000, 30.46667, 30.91389]

city_coords = dict(zip(cities_names, zip(x, y)))
```

Methodology

Genetic Algorithm Overview:

A Genetic Algorithm, inspired by the process of natural selection, was implemented to solve the TSP. The GA operates on a population of potential solutions (paths) and evolves them over multiple generations to find an optimal or near-optimal solution.

Key Components of the Genetic Algorithm

1. Representation: represented as a permutation of city names, indicating the order in which cities are visited.

2. Fitness Function: The fitness of a path is calculated as the total Euclidean distance traveled, including the return to the starting city. Lower fitness values indicate better solutions.

```
def euclidean_distance(city1, city2):
    return np.linalg.norm(np.array(city1) - np.array(city2))

def fitness(path, city_coords):
    total_distance = sum(euclidean_distance(city_coords[path[i]], city_coords[path[i+1]])
for i in range(len(path)-1))
    total_distance += euclidean_distance(city_coords[path[-1]], city_coords[path[0]])
    return total_distance
```

3. Selection: The population is sorted based on fitness, and the top performers are more likely to be selected for reproduction.

4. Crossover: A custom crossover operation is implemented to combine two parent paths and create a child path that maintains the validity of the TSP solution.

```
def crossover(parent1, parent2):
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child = [-1] * len(parent1)
    child[start:end] = parent1[start:end]
    pointer = 0
    for city in parent2:
        if city not in child:
            while child[pointer] != -1:
                pointer += 1
            child[pointer] = city
    return child
```

5. Mutation: Random swaps between cities in a path are performed to introduce diversity and explore new solutions.

```
def mutate(path, mutation_rate=0.01):
    for i in range(len(path)):
        if random.random() < mutation_rate:
            j = random.randint(0, len(path) - 1)
            path[i], path[j] = path[j], path[i]
    return path
```

6. Elitism: The top 10% of solutions from each generation are preserved to ensure the best solutions are not lost.

Algorithm Parameters

- Population Size: 250
- Crossover Rate: 0.8 (80% of new population created through crossover)
- Mutation Rate: 0.2 (20% of new population created through mutation)
- Number of Generations: 200

Implementation Details

The experiment was implemented in Python, utilizing libraries such as NumPy for numerical operations, Matplotlib and Seaborn for visualization, and built-in Python functions for randomization and list operations.
The main genetic algorithm function is implemented as follows:

```
def genetic_algorithm(city_coords, n_population, crossover_rate, mutation_rate,
n_generations):
    population = [random.sample(list(city_coords.keys()), len(city_coords)) for _ in
range(n_population)]

    for generation in range(n_generations):
        population = sorted(population, key=lambda path: fitness(path, city_coords))
        next_generation = population[:n_population // 10]  # Elitism

        for _ in range(int(n_population * crossover_rate)):
            parent1, parent2 = random.choices(population[:50], k=2)
            child = crossover(parent1, parent2)
            next_generation.append(child)

        for _ in range(int(n_population * mutation_rate)):
            mutated = mutate(random.choice(next_generation))
            next_generation.append(mutated)

        population = next_generation[:n_population]

        if generation % 10 == 0:
            best_path = population[0]
            print(f"Generation {generation} - Best fitness: {fitness(best_path,
city_coords):.2f}")

    return min(population, key=lambda path: fitness(path, city_coords))
```

Results and Discussion

The genetic algorithm was run for 200 generations, with the best fitness (shortest path length) reported every 10 generations. The algorithm consistently showed improvement over time, indicating successful evolution of solutions.
The final output of the algorithm provides:

1. The best path found (order of cities to visit).
2. A visualization of this path on a coordinate plane representing the geographical locations of the cities.

Performance Analysis

The genetic algorithm demonstrated effective optimization, consistently reducing the total path length over generations. However, due to the stochastic nature of genetic algorithms, running the experiment multiple times might yield slightly different results.

Visualization

The visualization component provides an intuitive representation of the best path found, plotting the cities on a 2D plane based on their longitude and latitude coordinates. This allows for easy interpretation of the solution and verification of its feasibility.
Here's the code for the visualization function:

```
def plot_path(city_coords, best_path):
    path_coords = [city_coords[city] for city in best_path] + [city_coords[best_path[0]]]
    x_coords, y_coords = zip(*path_coords)

    sns.set(style="whitegrid")
    plt.figure(figsize=(12, 8))
    plt.plot(x_coords, y_coords, "bo-", markersize=8, label="Path", alpha=0.7)

    for city, (x, y) in city_coords.items():
        plt.text(x, y, city, fontsize=12, ha="right", va="bottom")

    plt.title("Best Path Found by Genetic Algorithm", fontsize=16)
    plt.xlabel("Longitude", fontsize=14)
    plt.ylabel("Latitude", fontsize=14)
    plt.legend()
    plt.grid(True)
    plt.show()
```

Conclusion

This experiment successfully demonstrated the application of a genetic algorithm to solve the Traveling Salesman Problem for a set of Moroccan cities. The implemented algorithm showed consistent improvement over generations and provided a visually interpretable solution.
The project serves as a practical introduction to genetic algorithms and their application in solving complex optimization problems. It also highlights the interdisciplinary nature of such problems, combining elements of computer science, operations research, and geography.
To run the experiment, use the following code:

```
best_path = genetic_algorithm(city_coords, N_POPULATION, CROSSOVER_RATE, MUTATION_RATE,
N_GENERATIONS)
print(f"Best path found: {best_path}")
plot_path(city_coords, best_path)
```

This will execute the genetic algorithm and display the best path found, both as a list of cities and as a visual plot.