Project Work

Student Name: ABHISHEK KUMAR	**UID:**24MCC20006
Branch: MCA-CCD	**Section/Group:**1-A
Semester: 1st	**Date of Performance:30-10-24**
Subject Name: DESIGN AND ANALYSIS OF ALGORITHMS LAB	**Subject Code: 24CAP-612**

1.Aim:
The aim of this project is to implement and analyse different algorithms for solving the Traveling Salesman Problem (TSP). The TSP is a classic optimization problem in combinatorial optimization where the objective is to find the shortest possible route that visits each city exactly once and returns to the origin city. This project will explore various algorithmic approaches to solve the TSP and analyse their performance.

2.Task to be Done:

 Research TSP: Understand the problem definition, applications, and existing algorithmic approaches.

 Algorithm Selection: Choose several algorithms for solving TSP, such as:

- Brute Force
- Dynamic Programming (Held-Karp)
- Genetic Algorithm (heuristic approach)

 Implementation: Implement each selected algorithm in Python.

 Complexity Analysis: Analyze the time and space complexity of each algorithm.

 Performance Testing: Compare the performance of the algorithms using various datasets and visualize results.

 Documentation: Create comprehensive documentation of the project, including code comments, analysis, and findings.

3.Steps:

 Literature Review:

- Research the TSP problem, its significance, and the various approaches for solving it.
- Understand the mathematical formulation and graph representation of TSP.

 Select Algorithms:

- Decide on the algorithms to implement, considering both exact and approximate methods.

 Implementation:

- Write Python code for each selected algorithm.
- Ensure the code is modular and well-commented for clarity.

 Complexity Analysis:

- Determine the time and space complexity for each algorithm.
- Discuss the implications of these complexities on the performance with varying input sizes.

 Performance Testing:

- Create test cases with different numbers of cities.
- Measure and compare the execution time and efficiency of each algorithm.

 Documentation:

- Compile a report that includes the problem definition, algorithms used, performance results, and conclusions.

4.Code for experiment:

Using Brute Force Approach:-

```
import itertools

def calculate_distance(cities, path):
    distance = 0
    for i in range(len(path) - 1):
        distance += cities[path[i]][path[i + 1]]
    distance += cities[path[-1]][path[0]]  # Return to starting city
    return distance

def tsp_brute_force(cities):
    num_cities = len(cities)
    shortest_distance = float('inf')
    best_path = []

    for path in itertools.permutations(range(num_cities)):
        current_distance = calculate_distance(cities, path)
        if current_distance < shortest_distance:
            shortest_distance = current_distance
            best_path = path

    return best_path, shortest_distance
if __name__ == "__main__":
    cities = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]
# Brute Force
    best_path_brute_force, shortest_distance_brute_force = tsp_brute_force(cities)
    print("Brute Force Approach:")
    print(f"Best Path: {best_path_brute_force}, Shortest Distance: {shortest_distance_brute_force}")
```

Output:-

```
Brute Force Approach:
Best Path: (0, 1, 3, 2), Shortest Distance: 80
```

Using Dynamic Programming Approach:-

```python
def tsp_dynamic_programming(cities):
    n = len(cities)
    # memoization table
    memo = {}

    # helper function to find shortest path
    def find_shortest_path(pos, visited):
        if visited == (1 << n) - 1:  # All cities visited
            return cities[pos][0]  # Return to starting city

        if (pos, visited) in memo:
            return memo[(pos, visited)]

        min_cost = float('inf')

        for city in range(n):
            if (visited & (1 << city)) == 0:  # If city is not visited
                new_cost = cities[pos][city] + find_shortest_path(city, visited | (1 << city))
                min_cost = min(min_cost, new_cost)

        memo[(pos, visited)] = min_cost
        return min_cost

    return find_shortest_path(0, 1)  # Start from the first city with only the first city visited
if __name__ == "__main__":
    cities = [
        [0, 10, 15, 20],
        [10, 0, 35, 25],
        [15, 35, 0, 30],
        [20, 25, 30, 0]
    ]
# Dynamic Programming
    shortest_distance_dp = tsp_dynamic_programming(cities)
    print("Dynamic Programming Approach:")
    print(f"Shortest Distance: {shortest_distance_dp}")
```

Output:-

```
Dynamic Programming Approach:
Shortest Distance: 80
```

5.Conclusion:

The Traveling Salesman Problem (TSP) is a well-known combinatorial optimization problem that has significant implications in fields like logistics, planning, and circuit design. This project successfully implemented both brute force and dynamic programming approaches to solve TSP. The brute force method guarantees an optimal solution but is computationally expensive for larger datasets, while the dynamic programming approach significantly reduces computation time but is still limited by its complexity.

6.Learning Outcomes:

- **Understanding TSP**: Gained a deeper understanding of the Traveling Salesman Problem and its applications.
- **Algorithm Implementation**: Developed skills in implementing algorithms, both exact and approximate.
- **Complexity Analysis**: Learned to analyze and compare the time and space complexities of different algorithmic approaches.
- **Performance Testing**: Enhanced ability to conduct performance testing and analyze empirical results.
- **Critical Thinking**: Improved problem-solving skills through algorithm selection and performance optimization.

Teacher's Signature