Traveling Salesman Problem (TSP)

Report by: **Vivek Vittal Biragoni, 211AI041**
19/05/23

- TSP is a well-known combinatorial optimization problem.
- It involves finding the shortest possible route that visits a set of cities and returns to the starting city.
- TSP is NP-hard, meaning there is no known efficient algorithm to solve it for all cases.
- Various algorithms exist to approximate the solution, including exact algorithms for small instances and heuristic/metaheuristic approaches for larger instances.
- The problem is often represented using a graph, where cities are nodes and distances between cities are edge weights.
- The objective is to find a Hamiltonian cycle (a cycle that visits each city exactly once) with the minimum total distance.
- The brute-force approach for TSP involves examining all possible permutations of city visits, which becomes computationally infeasible as the number of cities increases.
- Approximation algorithms, such as the nearest neighbor algorithm or Christofides' algorithm, provide suboptimal but efficient solutions.
- Metaheuristic algorithms like simulated annealing, genetic algorithms, and ant colony optimization are commonly used to find good solutions for large instances.
- TSP has applications in logistics, transportation, scheduling, circuit design, and other domains requiring optimization of routes or tours.
- Researchers continue to develop new algorithms and techniques to improve the quality of solutions and handle larger TSP instances efficiently.

Problem Statement:

The problem is to solve the Traveling Salesman Problem (TSP) using the A* algorithm. Given a weighted adjacency matrix representing distances between cities, the task is to find the shortest path that visits all cities exactly once and returns to the starting city.

Data Structure Design

The code uses the following data structures:
1. Graph: It represents the graph of cities and their connections. The adjacency matrix is used to store the edge weights between cities.
2. State: It represents a state in the search space. It includes the current city, the visited cities, and the cost incurred so far.

3. Priority Queue: It is implemented using the `heapq` module and is used to prioritize the states based on their cost.

Algorithm Design

1. The code first constructs a graph using the provided adjacency matrix.
2. A minimum spanning tree (MST) is computed using Prim's algorithm to get an estimate of the minimum cost required to visit all remaining unvisited cities from the current city.
3. The A* algorithm is employed to explore the search space. The states are represented by the `State` class, which includes the current city, the visited cities, and the cost. The priority queue is used to prioritize the states based on their cost.
4. The algorithm starts from the initial state with the starting city and explores all possible next cities. It calculates the cost of reaching each unvisited city from the current city and adds new states to the priority queue.
5. The algorithm continues until all cities have been visited. The path with the minimum cost is returned as the solution.
6. Finally, the execution time of the algorithm is measured using the `timeit` module.

Sample Input and Output

Input:

- Weighted adjacency matrix:
  ```
  [[0, 15, 13, 16, 12],
   [15, 0, 14, 11, 17],
   [13, 14, 0, 19, 18],
   [16, 11, 19, 0, 14],
   [12, 17, 18, 14, 0]]
  ```
- City names: ["Arad", "Bucharest", "Craiova", "Dobreta", "Eforie"]

Output:

- Shortest path:
  ```
  ['Arad', 'Craiova', 'Dobreta', 'Bucharest', 'Eforie']
  ```

```
- Execution time: 0.00019210000755265355 seconds

## Output screenshot:


As per as the last part of the assignment i.e.

## **“Any other better search strategy for the problem.”**

Another jupyter file(or its converted pdf version) has been added by me in the zip file, in which I have explored three methods for the TSP.

Following is the gist of the additional work, the detailed work(which includes the notes, code for the tried out approaches and TSP) and detail can be found in the additional file

## Brute Force:

- Guarantees optimality by exhaustively searching all permutations.
- Time complexity grows factorially with the number of cities (O(n!)).
- Always finds the optimal solution but becomes impractical for large instances.

## Genetic Algorithm:

- Approximate solution approach for TSP.
- Time complexity depends on parameters and population size (O(num_generations * num_individuals * num_cities^2)).
- Solution quality depends on parameters and problem characteristics.
- Can handle larger instances compared to Brute Force, but finding high-quality solutions for very large problems can be challenging.
```

Held-Karp Algorithm:

- Guarantees optimality using dynamic programming.
- Exponential time complexity (O(2^n * n^2)).
- Provides the optimal solution with minimum distance and path.
- Practical for small to moderate-sized instances due to its time complexity.