Traveling Salesperson Problem (TSP)

BY: Vivek Vittal Biragoni, 211AI041 18/05/2023

```
import sys
import heapq
import timeit


class Graph:
    def __init__(self, num_vertices):
        self.num_vertices = num_vertices
        self.edges = [[0] * num_vertices for _ in range(num_vertices)]

    def add_edge(self, src, dest, weight):
        self.edges[src][dest] = weight
        self.edges[dest][src] = weight

    def minimum_spanning_tree(self):
        min_cost = [sys.maxsize] * self.num_vertices
        parent = [None] * self.num_vertices
        min_cost[0] = 0
        visited = [False] * self.num_vertices

        for _ in range(self.num_vertices):
            u = self._find_min_cost_vertex(min_cost, visited)
            visited[u] = True

            for v in range(self.num_vertices):
                if (
                    self.edges[u][v] > 0
                    and not visited[v]
                    and self.edges[u][v] < min_cost[v]
                ):
                    parent[v] = u
                    min_cost[v] = self.edges[u][v]

        return parent

    def _find_min_cost_vertex(self, min_cost, visited):
        min_value = sys.maxsize
        min_index = -1

        for v in range(self.num_vertices):
            if not visited[v] and min_cost[v] < min_value:
                min_value = min_cost[v]
                min_index = v

        return min_index


def tsp_a_star(adj_matrix, city_names):
    num_cities = len(adj_matrix)
    graph = Graph(num_cities)
    for i in range(num_cities):
        for j in range(num_cities):
            graph.add_edge(i, j, adj_matrix[i][j])

    start_city = 0
    mst_parent = graph.minimum_spanning_tree()

    def heuristic(state):
        remaining_cities = [city for city in range(num_cities) if city not in state.visited_cities]
        return mst_parent[state.current_city] + sum(graph.edges[state.current_city][city] for city in remaining_cities)

    class State:
        def __init__(self, current_city, visited_cities, cost):
            self.current_city = current_city
            self.visited_cities = visited_cities
            self.cost = cost

        def __lt__(self, other):
            return self.cost < other.cost
```

```
    initial_state = State(
        current_city=start_city,
        visited_cities=[start_city],
        cost=0
    )

    queue = []
    heapq.heappush(queue, initial_state)

    while queue:
        current_state = heapq.heappop(queue)

        if len(current_state.visited_cities) == num_cities:
            return current_state.visited_cities

        for next_city in range(num_cities):
            if next_city not in current_state.visited_cities:
                new_cost = (
                    current_state.cost + graph.edges[current_state.current_city][next_city]
                )
                new_visited_cities = current_state.visited_cities + [next_city]

                new_state = State(
                    current_city=next_city,
                    visited_cities=new_visited_cities,
                    cost=new_cost,
                )

                heapq.heappush(queue, new_state)

    return None


if __name__ == "__main__":
    # Weighted adjacency matrix for the graph
    adj_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"]

    # # Measure the execution time
    # start_time = time.time()
    path = tsp_a_star(adj_matrix, city_names)
    # end_time = time.time()

    if path:
        city_path = [city_names[city] for city in path]
        print("Shortest path:\n", city_path)
    else:
        print("No feasible solution found.")

    # # Print the execution time
    # execution_time = end_time - start_time
    # print("Execution time:", execution_time, "seconds")
    import timeit

# Function to measure the execution time
def measure_execution_time():
    # Weighted adjacency matrix for the graph
    adj_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"]

    # Solve TSP using A* algorithm
    path = tsp_a_star(adj_matrix, city_names)

# Measure the execution time
execution_time = timeit.timeit(measure_execution_time, number=1)

print("Execution time:", execution_time, "seconds")
```

```
Shortest path:
 ['Arad', 'Eforie', 'Dobreta', 'Bucharest', 'Craiova']
Execution time: 0.00011820002691820264 seconds
```

The code uses the following data structures:

1. **Graph**: The `Graph` class represents the graph of cities and their distances. It uses a 2D list (adjacency matrix) to store the distances between cities. The graph is initialized with the number of vertices (cities) and provides methods to add edges with their corresponding weights.
2. **Minimum Spanning Tree (MST)**: The `minimum_spanning_tree` method of the `Graph` class calculates the minimum spanning tree using Prim's algorithm. It utilizes arrays (`min_cost`, `parent`, and `visited`) to keep track of the minimum cost to reach each city, the parent city in the MST, and the visited status of each city.
3. **Priority Queue**: The `queue` variable is a priority queue implemented using the `heapq` module. It is used to store the states during the A* search. The states are organized based on their costs, with the state having the lowest cost at the top of the queue.
4. **State**: The `State` class represents a state in the search space. It contains information about the current city, the visited cities, and the cost associated with reaching the current state. The `__lt__` method is defined to compare states based on their costs, allowing them to be ordered in the priority queue.

The combination of these data structures enables efficient exploration of the search space, with the graph representing the cities and distances, the minimum spanning tree heuristic guiding the search, the priority queue managing the states based on their costs, and the state object encapsulating the necessary information for each state in the search.