2 The Travelling Trucker (54 marks)

In this programming question, you will solve a slightly modified instance of the Travelling Salesman Problem using the A* search and the breadth-first search algorithms, with multi-path pruning.

Suppose that you are a truck driver and your goal for the day is to deliver freight to all clients. You must leave the garage, make a stop at each client's warehouse to deliver freight, and return back to the garage at the end of the day. We relax the problem slightly so that you can drive through the same city more than once if needed. Since fuel is increasingly expensive, your company would like you to drive the minimum possible distance in total.

Given a list of locations with their respective latitudes and longitudes and the distances between connected cities, the task is to devise the shortest route that will take you from the garage, to each destination on the list, then back to the garage.

You are given two `.csv` files that contain all of the required information about the destinations and the map. Tables 2 and 3 illustrate the format of the `.csv` files for a map with 4 locations. Note that the `.csv` files do not contain column headers or row indices.

ID	Latitude	Longitude	Name
0	43.46	-80.52	Waterloo
1	43.45	-80.49	Kitchener
2	42.99	-81.25	London
3	42.40	-82.19	Chatham

Table 2: `locations.csv` gives the latitude, longitude, and name of the destinations. The row index gives the location ID.

	0	1	2	3
0	-1	3.6	-1	-1
1	3.6	-1	109	1
2	-1	109	-1	114
3	-1	-1	114	-1

Table 3: `distances.csv` gives the distance between locations on the map. The entry at row i and column j gives the distances between locations i and j. An entry of -1 indicates the locations are not directly connected by road.

The search problem is formally defined as follows:

- **States:** The state is the integer ID, latitude, and longitude of the current location, along with a set of the unvisited locations on the delivery list. For example, `(0, 43.46, -80.52, {1, 2})` indicates that the truck is in location 0 and has yet to deliver freight to locations 1 and 2. The state is implemented in the `State` class in the provided code.
- **Initial state:** The location of the garage and the set of locations to drop off freight
- **Goal state:** The location of the garage and the empty set of locations

- **Successor function:** State B is a successor of State A if and only if the following are true:
  - The location in state B is connected to state A by road
  - The location in state B is not equal to the location of state A
  - If State B's location is in the set of remaining locations in State A, then State B's set of remaining locations is the same as State A, but with State B's location removed. Otherwise, the set of remaining locations are equal for both states.
- **Cost function:** The road distance travelled between the locations in State A and State B

Information on the Provided Code

We have provided three Python files in a zipped folder on LEARN. Please read the detailed comments in the provided files carefully. Note that some functions have already been implemented for you. Your task is to implement all of the functions in `search.py`, except for `sample_heuristic()`.

1. `search.py`: Contains all the functions you will implement in this part of the assignment. You must implement the following functions: `is_goal()`, `get_path()`, `get_successors()`, `BFS()`, `A_star()`, and `custom_heuristic()`.
2. `utils.py`: Contains handy helper functions, along with implementations of a `State` class and a `Node` class for your search tree.
3. `example.py`: Demonstrates how to use the code to load the location and distance CSV files, and how to execute your search algorithms.

Zip and submit only the `search.py` file to Marmoset. We will use our version of `search.py` to test your code. Do not modify any provided function signatures in `search.py`. Doing so will cause you to fail our tests. Feel free to add any new code to `search.py`.

The Heuristic Functions for A* Search:

We have provided the implementation of a simple admissible heuristic function (see `sample_heuristic()` in `search.py`). This function is equal to the minimum road distance between any pair of locations multiplied by the number of remaining locations plus 1.

You must implement your own heuristic function in `custom_heuristic()`. As a hint, consider that the shortest travel distance between 2 points on Earth is the orthodromic distance. Consider using the `orthodromic_distance()` function available in `utils.py`.

Testing Your Program

Debugging and testing are essential skills for computer scientists. For this question, debugging your program may be especially challenging because of ties. Among "correct" implementations, the number of nodes expanded may vary widely due to how we handle the nodes with the same heuristic value on the frontier. Please test your code using **Python 3.8.5**.

Implement **multi-path pruning for both BFS and A***. When there are multiple paths to a state, multi-path pruning explores the first path to the state and discards any subsequent path to the same state. Use an explored set to keep track of the states that have been expanded by the algorithm. When you **remove** a state from the frontier, check whether the state is in the explored set or not. If the state is in the explored set, then do nothing. Otherwise, add the state to the explored set and continue with the algorithm. Note that we perform pruning after we **remove** a state from the frontier, not before we **add** a state to the frontier.

BFS's behaviour depends on the order of adding a state's successors to the frontier. We will break ties by using the number of remaining locations to visit in each state and their location IDs. Note that this will be done for you already if you produce a Python `set` of states with `get_successors()` and then sort it with `sorted()` (see the custom `==`, `<`, and `>` operators in the `State` class). At each step, BFS will add the successors to the frontier in **increasing** order of the number of remaining locations and then in increasing order of their IDs.

A* search will also break ties using the state comparison desribed above (see the overloaded operators for the `Node` class). Among several states with the same f-value, A* will expand the state with the fewest remaining locations. If two states have the same number of remaining locations, A* will break ties using the smallest location ID.

Please complete the following tasks:

Submit your solutions to part (a) on Marmoset and submit your solution to part (b) on Crowdmark.

(a) Complete the empty functions in `search.py` and submit `search.py` on Marmoset. Marmoset will evaluate your program for its correctness and efficiency.

For correctness, we have written unit tests for these functions: `is_goal()`, `get_path()`, `get_successors()`, `BFS()`, `A_star()`.

For each function, Marmoset provides one public test, which tests the function in a trivial scenario. There are also several secret tests. Before the deadline, you can only view the results of the public tests. After the deadline, Marmoset will run all the tests and calculate your marks.

Each test runs the function up to a predefined time limit. The test passes if and only

if the function terminates within the time limit and returns the expected result. Each test is all or nothing — there are no partial marks available.

Marking Scheme: (46 marks)

Unit tests on `get_path`, `is_goal`, `blocking_heuristic`, `get_successors`, `dfs`, and `a_star`.

- `is_goal()`: (1 public test + 2 secret tests) * 1 mark = 3 marks.
- `get_path()`: (1 public test + 2 secret tests) * 1 mark = 3 marks.
- `get_successors()`: (1 public test + 3 secret tests) * 2 marks = 8 marks.
- `BFS()`: (1 public test + 3 secret tests) * 4 marks = 16 marks.
- `A_star()`: (1 public test + 3 secret tests) * 4 marks = 16 marks.

Solutions: Submitted on Marmoset

(b) Describe the heuristic function you implemented in `custom_heuristic()`. In your answer, justify why it is consistent. Lastly, comment on how A* with your heuristic compares to A* with the sample heuristic (e.g., cost of solution, number of nodes expanded, domination). **Be as concise as possible**.

Marking Scheme: (8 marks)

- (2 marks) Clear description of custom heuristic
- (4 marks) Correct and concise justification for consistency
- (2 marks) Illustrative comparison with sample heuristic

Solutions: Heuristic implemented was a fairly basic one. It returns the inverse of (number of nodes away from initial node +1). custom heuristic shall be represented as $h_c(n)$ while sample shall be represented as $h_s(n)$ This is consistent as:

(a) The number of nodes never reduces the further away from the initial node.

(b) Therefore, the value of h(n) is always smaller than nodes before it for all n

(c) Thus, $h_c(n) \leq c(n, n') + h(n')$ for all n in the graph, and hence it is consistent.

Since the range of the custom heuristic is $0 < h_c(n) \leq 1$, for every node $h_s(n)$ wil be greater due to it using the actual distances + 1. Thus $h_s(n)$ dominates $h_c(n)$