Implement Route Planner

REVIEW | CODE REVIEW | HISTORY

Meets Specifications

The code works well. The answers are good.

Congratulations on successfully implementing A* search.

All the best!

Correctness

✓ **Running test.py shows "all tests pass".**

All tests pass! Congratulations!

✓ **The student implements all required methods.**

Your code works well.

In the future, consider using priority queue for the open set. Right now, your code takes `O(N^2)` time to run due to having to iterate over the frontier in search of the lowest-f node. It means that with a map of 1000 intersections (a rather small map in practice), your code needs to do 1 million calculations to find the optimal path, too slow for most practical purposes.

Using priority queue will reduce the runtime to `O(NlogN)`, so with 1000 intersections, we would only need to do roughly 10000 calculations.

Using priority queue is actually very easy. You only need to concern with 2 functions: `heappush` which push a new element to the queue, and `heappop` which removes and returns the smallest elements in the queue. You only need to modify 2-3 lines of your code.

To initialize an empty queue:

```
q = []
```

To add an element to the queue:

```
heapq.heappush(q, (priority, new_element))
```

To retrieve and remove the smallest element:

```
priority, best = heapq.heappop(q)
```

✓ **The heuristic function used to estimate the distance between two intersections is guaranteed to return a distance which is less than or equal to the true path length between the intersections.**

```
def distance(self, node_1, node_2):
    """ Computes the Euclidean L2 Distance"""
    # TODO: Compute and return the Euclidean L2 Distance
    x1, y1 = self.map.intersections[node_1]
    x2, y2 = self.map.intersections[node_2]
    return math.sqrt((x1-x2)**2 + (y1-y2)**2)
```

Euclidean distance, great choice for heuristics

✓ **Student answered all question correctly.**

All answers are good!

Choice and Usage of Data Structures

✓ **Code avoids obvious inappropriate use of lists and takes advantage of the performance improvement afforded by sets / dictionaries where appropriate. For example, a data structure like the "open_set" on which membership checks are frequently performed (e.g. `if node in open_set`) should not be a list.**

The use of data structure is generally appropriate.
In the future, for larger maps, consider using priority queue for the open set to improve the efficiency of the code further.

✓ **This item is a judgement call. Student code doesn't need to be perfect but it should avoid big performance degrading issues like...**

...unnecessary duplication of lists
...looping through a large set or dictionary when a single constant-time lookup is possible

No other issues

Some tips:

- Make use of built-in functions, such as `min` to shorten your code:

```
def get_current_node(self):
    """ Returns the node in the open set with the lowest value of f(node)."""
    # TODO: Return the node in the open set with the lowest value of f(node).
    current_node = None
    current_min = 0
    for node in self.openSet:
        if (current_node == None):
            current_node = node
            current_min = self.calculate_fscore(node)
        if self.calculate_fscore(node) < current_min:
            current_node = node
            current_min = self.calculate_fscore(node)
    return current_node
```

can be simplified to

```
def get_current_node(self):
    return min(self.openSet, key = lambda x: self.calculate_fscore(x))
```