Broad Institute Coding Challenge Design/Documentation

Candidate: Jeremy Leung

Documentation

Usage is very simple - the program is written in Python 3, so run it with:

```
python mbtaFunctions.py q1 q2 q3
```

Note that if Python 2 is tied to the python command by default, you would need to run it with python3 instead. The q1/q2/q3 arguments are optional, depending what output you want to see. For Q3 specifically, this runs a continuous loop that takes user input for two stations we want to return a route path for.

To run the test file, simply run:

```
python mbtaFunctions_test.py
```

Design decisions

Question 1

I first called the “/routes” GET endpoint to get a list of routes to answer Question 1. This was pretty straightforward, and I used the suggested approach of filtering with GET params (for route type 0 and 1). While this does rely on the API to perform the filtering logic, which could change and would require us to call the API again if we decide to get all route types instead, it ends up saving the amount of data transferred over the API call, as well as reduces the memory needed to store the JSON response we receive.

Question 2

I then called the “/stops” GET endpoint to get stops for each route. I originally wanted to only call the API endpoint once for **all** routes, but including the route IDs as a comma separated list in the filter parameter gives us a single large list of all the stops that match these routes. I wanted to then split these stops up by route, but adding “include=route” didn’t work, since the API filled the “relationships”->”route” field with ‘{“data”:null}’ if you include multiple routes. Thus, I need to call this stops endpoint once for each route received in the first response. If I don’t lump the routes together for this API call, the “relationships”->”route” field is appropriately populated with the route given to the endpoint, so it must be something with MBTA’s implementation where they don’t return multiple routes in that field.

This makes it easy to answer the beginning of the second question - for each route, call the API, and count the number of stops in the response. We track the least/most number of stops and its corresponding route name and print that. There could be multiple routes with the same number of stops - in this case we want to print all of these routes (which applies since Green Line D/E both had the most stops with 25).

For question 2.3, we want to know subway stops that touch multiple routes - the easiest way to track this would be to do a subway stop - subway routes mapping, and return all the ones that have more than one route mapped. The cleanest way to do this in one pass is to iterate through all the stops returned while going through each route, and if a stop already has an entry in this map, we add it to a list of stops with multiple routes, and print the map/dictionary entries for the stops with multiple routes in the response (which is a list of route names).

Question 3

My initial thought was to make a graph representation of the whole MBTA system, but I realized that the question just asked for the route names needed for this path. This meant that as long as there was a way to get from one route to another that eventually led to the destination, the question is sufficiently answered, meaning I just need to track the route names and the other lines they touch (meaning each route is a node and its intersecting lines are neighbors). Using the info about stations that touch multiple routes from question 2.3, we can then convert this to a route to connections mapping, for example, Park Street would tell us Red Line connects to Green B/C/D/E, and Ashmont would tell us Red Line connects to Mattapan Trolley.

The algorithm to find the route is simple, and likely not optimal, but answers the question appropriately. We basically do DFS, and return the first path that has the least route transfers. For example, for Harvard -> Aquarium the program would return [‘Red Line’, ‘Green Line B’, ‘Blue Line’] (Green Line C/D/E also work but we just return the first one by MBTA’s sorting of routes if all else is equal, which is Red > Mattapan > Orange > Green B/C/D/E > Blue, as shown in Q1).

We take user input as a comma separated list of the two stations, and do basic sanitation of input and stripping of beginning/ending spaces. We then look through each route that the source/destination stations are connected to, and see if there exists a path between these two routes that is shorter than the current shortest one. So for example, Park Street to Harvard could have multiple possible answers like [‘Red Line’, ‘Green Line B’, ‘Green Line C’, ‘Red Line’] or [‘Red Line’], and the latter would be the shortest path.

Taking continuous input

The decision to have the program run continuously and await for user input was because constructing the stops to routes and route to connections mapping was expensive, and the third question is likely one that the user would want to test with a variety of inputs. We don’t want to

call the MBTA API each time (we would also get rate limited, as I ran into a couple times during development), and we don't want to reconstruct these mappings every time as well.

In a production environment, we would presumably have these maps stored in a database somewhere instead of constantly reconstructing them, and only run the reconstruction when we know the MBTA subway map has been updated, or there are station closures or whatnot.

Other remarks

Note that our data structure and algorithm for Question 3 is very much geared towards answering what the question is asking specifically. If we want to do actual route finding between two stops that include interchange stations (i.e. Harvard -> Aquarium, change at Park Street to Green Line B, then change at Gov't Center to Blue Line), then we would need an actual graph representation of the subway system. Our solution also doesn't return the shortest path in terms of number of stops; for example, going from Ashmont to Aquarium would tell the user to go through the Green Line instead of Orange Line even though changing at Chinatown to the Orange Line would be quicker, since I went with simplicity for the implementation and returned the first shortest path. This may make extending the model to account for follow up questions a little more challenging as well.

Testing

I added unit tests for all functions in mbtaFunctions.py, which in turn tests the questions asked in Q1/Q2/Q3 along with helper functions. Most tests were pretty straightforward - couple key things to call out here would be determining test cases for Q3. I included tests for the most possible route transfers (Mattapan - Blue Line, which would need 4 total routes), and the least transfers (any two stations along the same line), as well as least transfers where both stations have multiple intersecting routes (i.e. Park Street, Arlington, Boylston, etc.). I also included some spot checks for other route combinations for testing Q3. The unit tests definitely aren't exhaustive for some of the test functions but should cover most of our bases.