Problem Set 2: Fastest Way to Get Around Boston

Handed out: Monday October 29, 2018
Pset Due: 9pm on Wednesday, November 7, 2018
Checkoffs: Monday November 12th - Monday November 26th

Getting Started:

Download files:

1. `ps2.py`: code skeleton
2. `graph.py`: a set of graph-related data structures (Digraph, Node, and Edge) that you must use
3. `t_map.txt`: a data file holding information about the Boston T map

Introduction:

In this problem set you will solve a simple optimization problem on a graph. Specifically, you will find the quickest route from one T-stop to another, given that you only want to take certain color T-lines.

Figure 1. Boston T Map

Here is the map of the Boston T. For simplification, in our text file, the only branch of the green line that is included is the E branch (to HEATH) . The rest of the lines are included as shown in the image, and some bus routes are included as the 'purple' line.

The T Text file format:

Each line in t_map.txt has 4 pieces of data in it in the following order separated by a single space:

1) The `start` T-stop (stop names do not contain spaces, spaces are denoted by underscores like: kendall_square)
2) The `destination` T-stop
3) The `time` in minutes to get between those two stops
4) The `color` of the T line that contains this path

For example:

```
kendall_square central 3 orange
prudential symphony 4 green
```

The text file contains a line with this data for each pair of **adjacent** stops on the T. Remember that the T travels in both directions, and a directed edge will need to be created for both directions between two stops.

However, the text file only includes each pair of adjacent stops once.

Problem 1: Creating the Data Structure Representation

In `graph.py`, you'll find the `Node` class, which has already been implemented for you.

You will also find skeletons of the `WeightedEdge` and `Digraph` classes, which we will use in the rest of this problem set.

Complete the `WeightedEdge` and `Digraph` classes such that the unit tests in the tester.py file for `WeightedEdge` and `Digraph` pass. Your `WeightedEdge` class will need to implement the `__str__` method (which is called when we use `str()` on a `WeightedEdge` object) as follows:

> Suppose we have a `WeightedEdge` object `e` containing by the following information:
>
> Source node name: `'a'`
> Destination node name: `'b'`
> Time in minutes along the edge: `3`
> Color of the t line: `'orange'`
>
> Then `str(e)` with the above information should yield:
> `a -> b 3 orange`

For Digraph, you will need to implement the `get_edges_for_node`, `has_node`, `add_node`, `add_edge` methods.

Note 1: In the add_node() method, you have to add a node to the self.nodes attribute of Digraph. Notice that we initialize self.nodes as a set() object. In python, a set is an unordered group of **unique** elements, meaning if you try to add an element that is already in the set, it will still only occur in the set once. Adding an element to a set can be done by using the set.add(element) method. (This is the only thing we will need sets for in this pset, but to read more about sets, you can look at this tutorial or the python set documentation.)

Note 2: All edge weights should be stored as **integers.**

Problem 2: Building up the T Map

For this problem, you will be implementing the `load_map(map_filename)` function in `ps2.py`, which reads in data from a file and builds a directed graph to properly represent the T map. Think about how you plan on representing your graph before implementing `load_map`.

Problem 2a: Designing your graph:

Decide how the T map problem can be modeled as a graph. Be prepared to explain yourself during your checkoff. (You can write down notes as comments in your code if you want.) *What do the graph's nodes represent in this problem? What do the graph's edges represent in this problem? Where are the distances represented?*

Problem 2b: Implementing load_map:

Implement `load_map` according to the specifications provided in the docstring. You may find this [link]() useful if you need help with reading files in Python (refer to section 7.2).

Problem 2c: Testing load_map:

Test whether your implementation of `load_map` is correct by creating a text file, `test_load_map.txt`, using the same format as ours (t_map.txt), loading your txt file using your `load_map` function, and checking to see if your directed graph has the correct nodes and edges. We have already implemented the __str__ method for a digraph, so you can print a digraph to see which edges it contains. You can add your call to `load_map` directly below where `load_map` is defined, and comment out the line when done. You will be asked to demonstrate this during your checkoff. Your test case should have at least 3 nodes and 3 lines. For example, if your test_load_map.txt was:

```
a b 3 red
a c 2 green
b c 4 red
```

Then your load_map function would return a digraph with 6 edges (in any order):

```
a -> b 3 red
b -> a 3 red
a -> c 2 green
a -> c 2 green
b -> c 4 red
c -> b 4 red
```

Submit `test_load_map.txt` when you submit your pset. Also, include the lines used to test `load_map` at the location specified in `ps2.py`, but **comment them out**.

Problem 3: Shortest Path using Optimized Depth First Search

We can define a valid path from a given start to end node in a graph as an ordered sequence of nodes [n_1, n_2, ... n_k], where n_1 to n_k are existing nodes in the graph and there is an edge from n_i to n_{i+1} for i=1 to k - 1. In Figure 2, each edge is unweighted, so you can assume that each edge has distance 1, and then the total distance traveled on the path is 4.

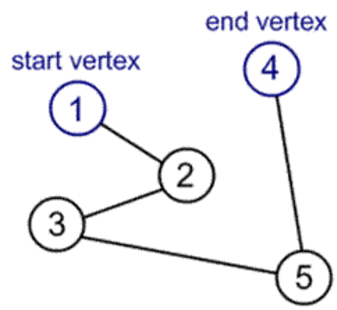

Figure 2. Example of a path from start to end node.

In our T map problem, the **total time traveled** on a path is equal to the sum of all time traveled between adjacent nodes on this path. Depending on the number of nodes and edges in a graph, there can be multiple valid paths from one node to another, which may consist of varying time-durations. We define the **shortest path** between two nodes to be the path with the **least total time spent travelling**. You are trying to minimize the time traveled while **not passing through certain color lines on the T**.

How do we find a path in the graph? Work off the depth-first traversal algorithm covered in lecture to discover each of the nodes and their children nodes to build up possible paths. Note that you'll have to adapt the algorithm to fit this problem. You can read more about depth-first search [here](#).

Problem 3a: Objective function

What is the objective function for this problem? What are the constraints? Be prepared to talk about the answers to these questions in your checkoff.

Problem 3b: Implement add_node_to_path

Implement the helper function `add_node_to_path`, as described in the docstring. Be sure you pass the test for `add_node_to_path`. We recommend using this function to add nodes to your path when implementing `get_best_path` in order to make sure you do not unintentionally mutate the path.

Problem 3c: Implement get_best_path

Implement the helper function `get_best_path`. Assume that any variables you need have been set correctly in `directed_dfs`. Below is some pseudocode to help get you started.

```
if start and end are not valid nodes:
        raise an error
elif start and end are the same node:
        update the appropriate variables
else:
        for all the child nodes of start
                construct a path including that node
                recursively solve the rest of the path, from the child node to the end node

return the shortest path
```

When you run tester.py, below the lines that say whether you failed or passed the tests, we also print more details about what paths we are testing and the **expected vs your** output for the paths. This may help you with debugging.

Notes:

1. Graphs can contain cycles. A cycle occurs in a graph if the path of nodes leads you back to a node that was already visited in the path. When building up possible paths, if you reach a cycle without knowing it, you could get stuck indefinitely by extending the path with the same nodes that have already been added to the path.

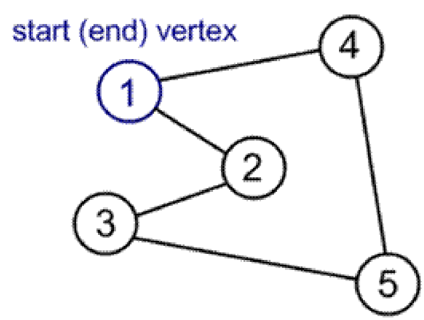

Figure 3. Example of a cycle in a graph.

2. If you come across a path that is longer than your shortest path found so far, then you know that this longer path cannot be your solution, so there is no point in continuing to traverse its children and discover all paths that contain this sub-path. You must include this optimization in your solution in order to receive full credit.
3. While not required, we strongly recommend that you use recursion to solve this problem.

If you would like to, you can uncomment the lines at the bottom of ps2.py and change the values of start,end,restricted_colors in order to debug get_best_path and see what your get_best_path will return.

Problem 4: Using directed dfs:

The function `directed_dfs(digraph, start, end, restricted_colors)` uses this optimized depth first search to find the **shortest path** in a directed graph from start node to end node under the following constraint: you do not pass on any edges from the colors listed in `restricted_colors`. Read directed DFS and understand how it uses the helper function get_best_path.

You do not have to implement anything for this part, but be sure you know what directed_dfs is doing, and can explain it during a checkoff.

Hand-In Procedure

1. Save

Save your solutions as **`graph.py`**, **`ps2.py`**, and **`test_load_map.txt`**

2. Time and Collaboration Info

At the start of each file, in a comment, write down the number of hours (roughly) you spent on the problems in that part, and the names of the people you collaborated with. For example:

```
# 6.0002 Problem Set 2
# Name: Jane Lee
# Collaborators: John Doe
# Time:
#
... your code goes here ...
```

3. Sanity checks

After you are done with the problem set, do sanity checks. Run the code and make sure it can be run without errors and passes our test cases as well as your own.

4. Submit

Upload all your files to the [6.00 submission site](). If there is an error uploading, email the file to 6.0002-staff@mit.edu.

You may upload new versions of each file until the 9:00PM deadline, but anything uploaded after that will be ignored, unless you still have enough late days left. We will use your last submission to grade and base late days on.