Problem Set 2: Routing and Path Optimization

Handed out: Monday, October 28, 2019
Due: Monday, November 4th, 2019 at 9PM
Checkoffs: Thursday, November 7th 2019 to **Friday, November 15th, 2019 at 5PM**

Objectives

- Create data structure representations of directed graphs
- Use Dijkstra's algorithm to find the best path between two nodes on a graph
- Apply computing and data analysis to an urban studies and planning problem

Collaboration

- Students may work together, but students are **not permitted to look at or copy each other's code or code structure.**
- Each student should write up and hand in their assignment separately.
- Include the names of your collaborators in a comment at the start of each file.
- **Please refer to the collaboration policy in the Course Information for more details.**

Introduction

MIT's Urban Science Department has a project where they want students to help implement an algorithm to find the optimal path for traveling between two nodes on a graph. You, after learning about searching and pathfinding in 6.0002, decide to take on the task. Specifically, for the project you will find the "best" driving route from home (**N0**) to work (**N9**) given a set of conditions, and help other drivers find the "best" driving route between other nodes.

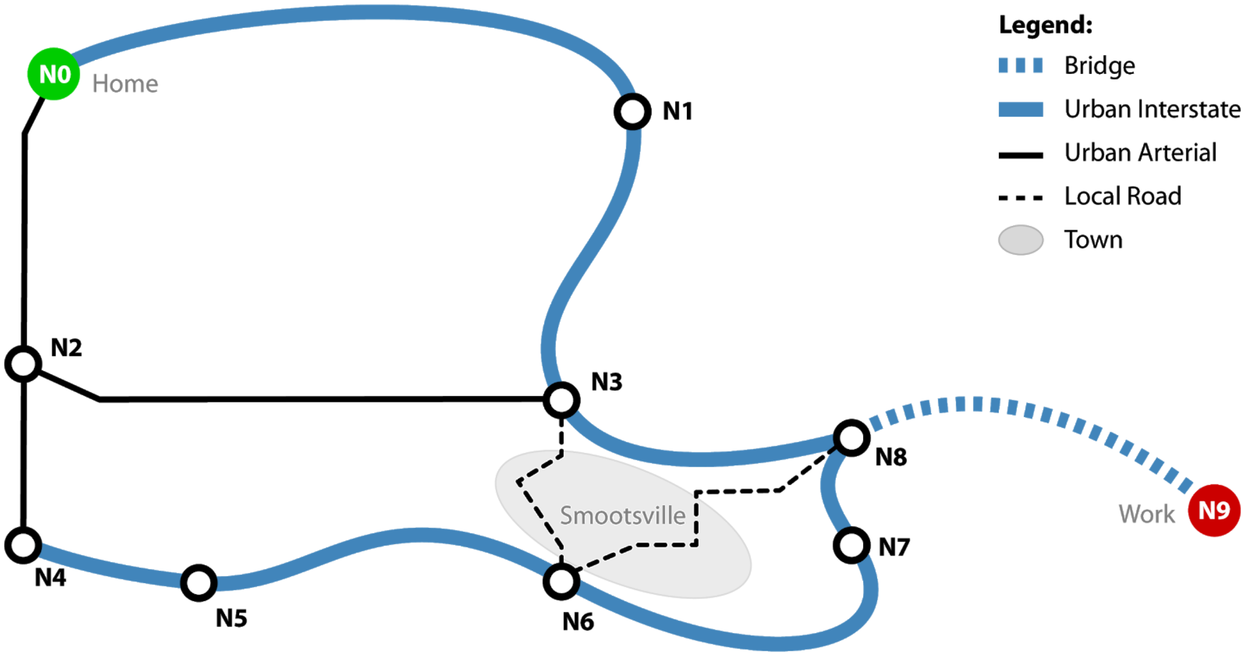

Figure 1. Road Map

Here is the map of our road network. The map has 10 nodes (`"N0"`, `"N1"`, … , `"N8"`, and `"N9"`) and has 13 roads of various types. The roads include some local roads that pass through the quiet town of Smootsville, as well as interstate roads.

Getting Started

Download **ps2.zip** from the problem set website. This folder contains the following files:

- **ps2.py**: code skeleton
- **graph.py**: a set of graph-related data structures (RoadMap, Node, and DirectedRoad) that you must use
- **maps/road_map.txt**: a data file holding information about the road network
- **ps2_tester.py**: basic tests for your code

Make sure all these files are saved in the same directory. Please do not rename these files, change any of the provided helper functions, change function names, or delete docstrings.

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 `DirectedRoad` and `RoadMap` classes, which we will use in the rest of this problem set.

Complete the `DirectedRoad` and `RoadMap` classes such that the unit tests in the `ps2_tester.py` file for `DirectedRoad` and `RoadMap` pass. Your `DirectedRoad` class will need to implement the `__str__` method (which is called when we use `str()` on a `DirectedRoad` object) as follows:

Suppose we have a `DirectedRoad` object `o` representing the following information:

Source node name: '`N0`'
Destination node name: '`N1`'
Time in minutes along the road: `10`
Type of road: '`interstate`'

Then `str(o)` should return:

```
N0 -> N1 takes 10 hours via interstate road
```

For `RoadMap`, you will need to implement the `get_roads_for_node`, `has_node`, `add_node`, `add_road` methods.

Note 1: In the `add_node()` method, you have to add a node to the `self.nodes` attribute of `RoadMap`. Note 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 appear 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 road travel times should be stored as integers.

Problem 2: Building the Road Network

For this problem, you will implement the `load_map(map_filename, is_normal_time)` function in `ps2.py`, which reads data from a file and builds a directed graph to represent the road network. Think about how you will represent your graph before implementing `load_map`.

Part 2A: Designing your graph

Decide how the road network problem can be modeled as a graph. Be prepared to explain yourself during your checkoff. (You may 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? How are the travel times represented?

Part 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.

Road Network Text File Format

Each line in `road_map.txt` has 5 pieces of data in the following order separated by a single space:

1. The `start` node
2. The `destination` node
3. The `time` in minutes to go between the two nodes
4. The `type` of road connecting these two nodes

For example:

```
N0 N1 15 interstate
N1 N3 6 interstate
N3 N6 5 local
```

The text file contains one line with this data for each pair of adjacent nodes in our network. **However, the text file only includes each pair of adjacent nodes once. All roads can be traversed in both directions, so a DirectedRoad object will need to be created for each direction between two nodes.**

Part 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 (`road_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 `RoadMap`, so you can print a `RoadMap` object to see which roads 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 roads, and should be **different** from the example `test_load_map.txt` below and `road_map.txt`.

For example, if your `test_load_map.txt` was:

```
a b 3 local
a c 6 interstate
b c 6 bridge
```

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

```
a -> b takes 3 hours via local road
b -> a takes 3 hours via local road
a -> c takes 2 hours via interstate road
c -> a takes 2 hours via interstate road
b -> c takes 4 hours via bridge road
c -> b takes 4 hours via bridge road
```

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

Problem 3: Shortest Path Using Dijkstra's Algorithm

We can define a valid path from a given start to end node in a graph as an ordered sequence of nodes [$n_1, n_2, \ldots n_k$] , 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. Thus, the total distance traveled on the path is 4.

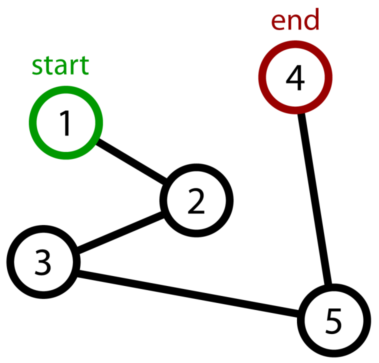

Figure 2. Example of a path from start to end node.

In our routing problem, the **total time traveled** on a path is equal to the sum of the times traveled between adjacent nodes on this path. Note that there may be multiple valid paths from one node to another with different total times traveled. We define the **shortest path** between two nodes to be the path with the **least total time spent travelling**.

How do we find a path in the graph? Work off Dijkstra's algorithm covered in lecture to traverse each node and build up possible paths. Note that you will have to adapt the algorithm to fit this problem. You can read more about Dijkstra's algorithm here.

Part 3A: Objective function

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

Part 3B: Implement get_neighbors

As a warmup for working with our representation of graphs, implement the function `get_neighbors`, which takes a RoadMap object, a source node, and a list of restricted road types and returns a list of nodes that are connected by directed edges emanating from the source node. *Hint:* you may find the `get_roads_for_node` function in the RoadMap object helpful.

Part 3C: Implement get_best_path

Implement the function `get_best_path`. This function uses Dijkstra's algorithm to find the **shortest path** in a directed graph from the **start** node to the **end** node under the following constraint: you do not pass on any roads from the types listed in `restricted_roads`. The function then returns the **(shortest path, shortest time)** tuple. Below is some pseudocode to help get you started:

```
function get_best_path(graph):
    if either start or end is not a valid node:
        return None
    if start and end are the same node:
        return ([], 0) # Empty path with 0 travel time

    Label every node as unvisited.

    Label every node with a shortest time value from the start
    node, with the start node being assigned a travel time of 0 and
    every other node assigned a travel time of ∞.

    # Repeat from here later on
    while there are unvisited nodes:
        Set unvisited node with least travel time as current node.

        For the current node, consider each of its unvisited
        neighbors, and (if necessary) update the best time and best
        path from the start node to these neighbors.

        Mark the current node as visited.

    if end node is not marked visited:
        No path exists between start and end. Return None.

    return (best path, best time) from start to end
```

Note: because this is pseudocode, you will have to choose some concrete implementation details (e.g. data structures used, base cases) for yourself.

When you run `ps2_tester.py`, below the lines that say whether you failed or passed the tests, we also print more details about which path we are testing and how your output differs from the expected output. This may help you with debugging.

Notes:

1. Graphs can contain cycles. A cycle occurs in a graph if a path visits the same node more than once. When building up possible paths, if you reach a cycle without knowing it, you could create an infinite loop by including a node more than once in a path. Make sure your code avoids exploring paths containing such cycles.

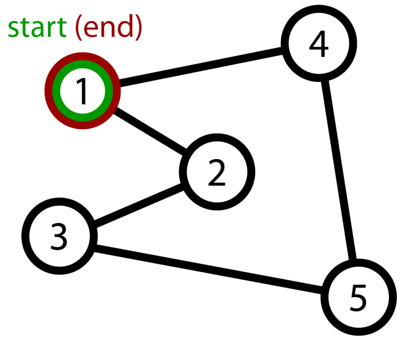

Figure 3. Example of a cycle in a graph.

2. **You must use Dijkstra's algorithm for `get_best_path` or an optimized version of DFS/BFS or your code will time out and fail our hidden tests.**

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

Problem 4: Using the best path algorithm

Now that you have a working `get_best_path` algorithm, let's call it to answer some questions:

Part 4A: Shortest Path With No Traffic

Implement `best_path_ideal_traffic`, which calculates the shortest path from **start** to **end** during **normal** traffic conditions. Your function should return a list of the nodes of the best route.

Part 4B: Shortest Path With Restricted Roads

The town of Smootsville, drowning in congestion, has decided to impose a ban on non-resident commuters passing through their town (this actually happens). Implement `best_path_restricted`, which calculates the new shortest path from **start** to **end** that **does not use local roads**.

Part 4C: Shortest Path With Restricted Local Roads To A Nearby Transit Stop

In order to further alleviate traffic, the city is encouraging commuters to park in the nearest neighboring town to their destination and take public transit to their final stop.

I. Complete your implementation of `get_best_path` to make use of the `to_neighbor` parameter while calculating the shortest path. That is, in the case that `to_neighbor = True`, should return the shortest path from **start** to **the neighbor of end that is closest to start**. This may require some refactoring of the code that you wrote in **Part 3B**. *Hint*: you may find `get_neighbors` from **Part 3A** to be useful. Note: `to_neighbor` is a parameter with a default value of `False`, which means that it is automatically assigned a value of `False` if it is not specified during a call to `get_best_path`; if you are confused about this syntax, read more here.

II. Implement `best_path_to_neighbor_restricted`, which calculates the new shortest path from **start** to **the neighbor of end that is closest to start** that **does not use local roads**.

Further Considerations

Think about the following questions, which you might naturally ask while completing this PSET:

- Do you think it is fair to prohibit non-resident commuters from driving through the local town streets? If so, under what conditions should this be implemented? What about the fairness of charging tolls on one or more of the roads?

- How might you use your graph to test other interventions that might help various parties? Note also that travel times will depend on road congestion, so more modeling and analysis is needed to evaluate the equilibrium conditions that would result if each traveler chose their best path to work based on the congestion created by other travelers.

- If a fraction of commuters used the city streets and the rest took the highway, the total travel time for all commuters might in fact be less than if all commuters took—or avoided—the local streets. Is it okay to give realtime 'shortcut' information only to subscribers who pay for premium services such as Waze? Which policies and regulations might lead to a reasonably efficient situation and while still being considered fair by all parties?

MIT Course 11+6 (Urban Science and Planning with Computer Science)

This problem set was made in partnership with Course 11+6, which offers a Bachelor of Science in Urban Science and Planning with Computer Science.

Do you like these sorts of problems involving both significant knowledge of computing and data analysis, and a deeper understanding of social choice, governance, urban planning, and regulation?

Visit MIT Course 11-6: Urban Science and Planning with Computer Science or email urban-science@mit.edu to speak with an advisor.

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 Python 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. **Be sure to run ps2_tester.py and make sure all the tests pass.**

Note: Passing all of the tests in **ps2_tester.py** does not necessarily mean you will receive a perfect score on the problem set. The staff will run additional tests on your code to check for correctness.

4. Submit

Upload **graph.py, ps2.py**, and **test_load_map.txt** to the 6.0002 submission site. If there is an error uploading, attach your files to a private Piazza post. **DO NOT upload any other files other than the three listed.**

You may upload new versions of each file until the **9PM** deadline, but anything uploaded after that will get a score of 0, unless you have enough late days left.

After you submit, please make sure the files you have submitted show up on the problem set submission page. When you upload a new file with the same name, your old one will be overwritten.