Problem Set 2: Fastest Way to Get Around MIT

Released: Monday, October 31, 2016.
Due: 11:59pm, Wednesday, November 9, 2016.

Introduction

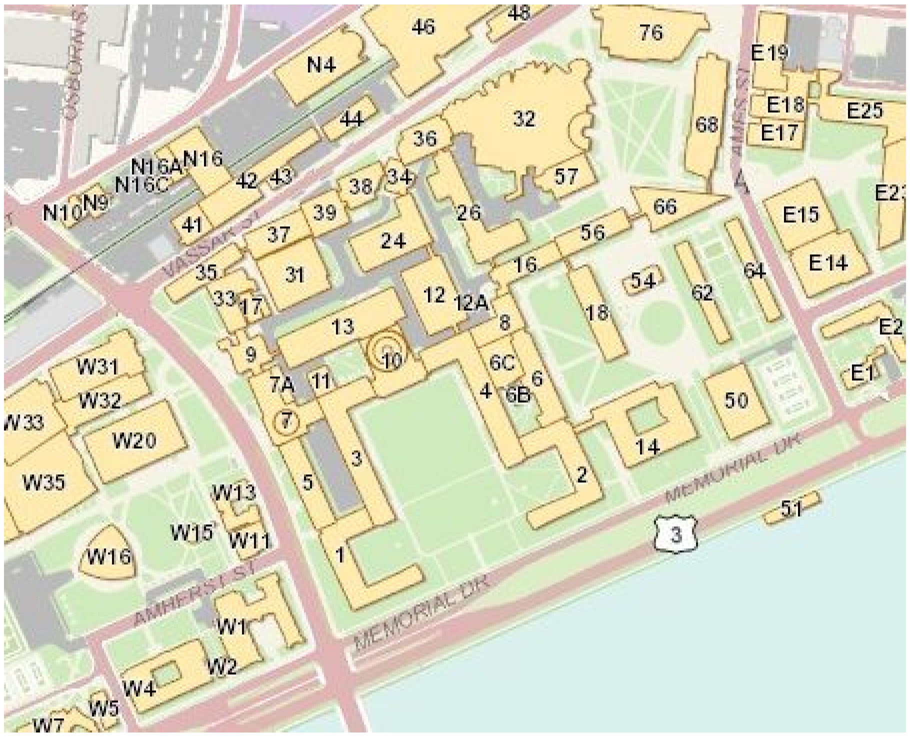
In this problem set you will solve a simple optimization problem on a graph. Specifically, you will find the shortest route from one building to another on the MIT campus given that you wish to constrain the amount of time you spend walking outdoors (in the cold).

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. `mit_map.txt`: a sample data file that holds the information about an MIT campus map.

Introduction



Here is the map of the MIT campus that we all know and love. From the text input file, `mit_map.txt`, you will build a representation of this map in Python using the graph-related data structures that we provide.

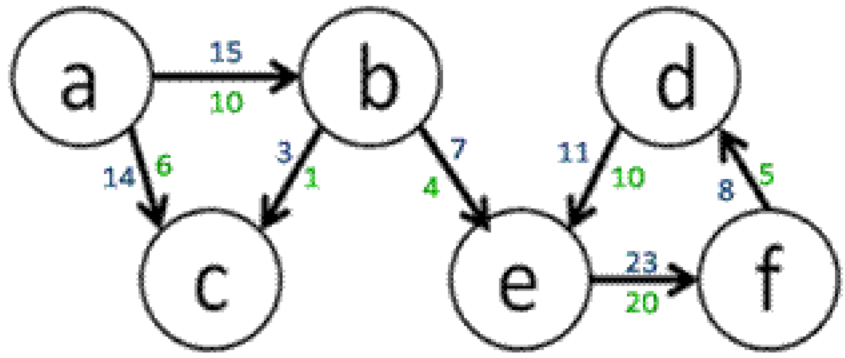
Each line in `mit_map.txt` has 4 pieces of data in it in the following order separated by a single space (space-delimited): the start building, the destination building, the distance in meters between the two buildings, and the distance in meters between the two buildings that must be spent outdoors. For example, suppose the map text file contained the following line:

```
10    32    200    40
```

This means that the map contains an edge from building 10 (start location) to building 32 (end location) that is 200 meters long, where 40 of those 200 meters are spent outside. During the day, routes are crowded and only allow for you to travel in one direction.

Problem 1: Creating the Data Structure Representation

In this problem set, we are dealing with edges that have different weights. In the figure below, the blue numbers show the cost of traversing an edge in terms of total distance traveled, while the green numbers show the cost of traversing an edge in terms of distance spent outdoors. Note that the distance spent outdoors for a single edge is always less than or equal to the total distance it takes to traverse that edge. Now the cost of going from “a” to “b” to “e” is a total distance traveled of 22 meters, where 14 of those meters are spent outdoors.



In `graph.py`, you’ll find the Node, and Edge classes, which do not store information about weights associated with each edge. 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 at the bottom of `graph.py` 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'
Total distance along the edge: 15
Outdoor distance along the edge: 10
Then `str(e)` with the above information should yield:
a->b (15, 10)

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

Problem 2: Building up the Campus 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 MIT campus map according to the data. Think about how you plan on representing your graph before implementing `load_map`.

Problem 2a: Designing your graph

Decide how the campus map problem can be modeled as a graph. Write a description of your design approach as a comment under the Problem #2 heading in `ps2.py`. 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 2a: Designing your graph
#
# What do the graph's nodes represent in this problem? What
# do the graph's edges represent? Where are the distances
# represented?
#
```

Problem 2b: Implementing load_map

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

https://docs.python.org/3/tutorial/inputoutput.html

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, loading your txt file using your load_map function, and checking to see if your directed graph has the nodes and edges it should. You can add your call to load_map directly below where load_map is defined, and comment out the line when you're done testing (It may also help to comment out the __main__ code block to clean up your output while testing this function). Your test case should have at least 3 nodes and 3 edges. For example, if you had Nodes "a", "b", and "c" and edges `WeightedEdge(a, b, 10, 9)`, `WeightedEdge(a, c, 12, 2)`, and `WeightedEdge(b, c, 1, 1)`, if you were to print out your graph, you would see something like:

```
Loading map from file...
a->b (10, 9)
a->c (12, 2)
b->c (1, 1)
```

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

Problem 3: Find the 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 $[n_1, n_2, \ldots n_k]$, 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, and then the total distance traveled on the path is 4.

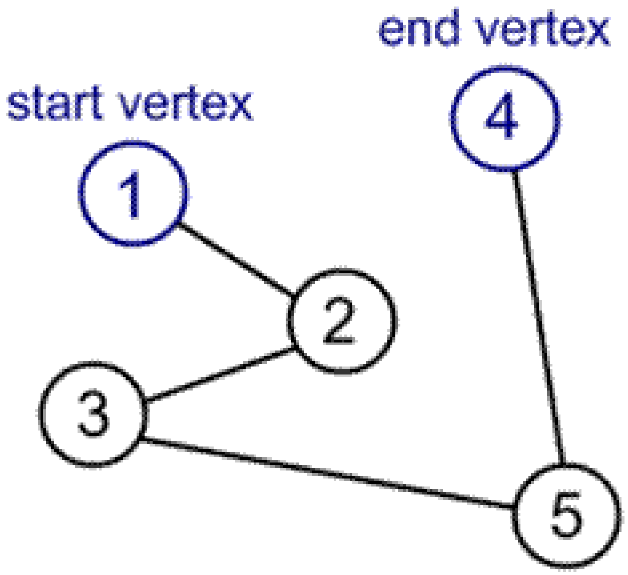


Figure 2. Example of a path from start to end node.

In our campus map problem, the **total distance traveled** on a path is equal to the sum of all total distances traveled between adjacent nodes on this path. Similarly, the **distance spent outdoors** on the path is equal to the sum of all distances spent outdoors on the edges in the 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 distances. We define the **shortest path** between two nodes to be the path with the **least total distance traveled**. You are trying to minimize the distance traveled while not exceeding the maximum distance outdoors.

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. Read more about depth-first search here.

Problem 3a: Objective function

Write a sentence describing what is the objective function for this problem.

```
# Problem 3a: Objective function
#
# What is the objective function for this problem? What are the constraints?
#
```

Problem 3b: 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 global variables appropriately
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
```

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.

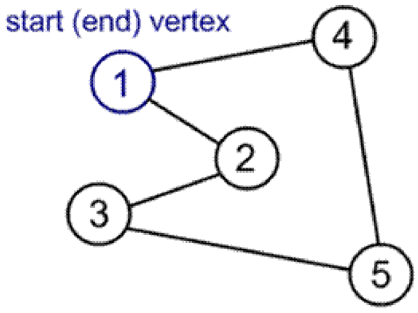


Figure 3. Example of a cycle in a graph.

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.

Problem 3c: Implement directed_dfs

Implement the function `directed_dfs(digraph, start, end, max_total_dist, max_dist_outdoors)` that uses this optimized depth first search to find the **shortest path** in a directed graph from start node to end node under the following constraints: the total distance travelled is less than or equal to `max_total_dist`, and the total distance spent outdoors is less than or equal to `max_dist_outdoors`. If multiple paths are still found, then return any one of them. If no path can be found to satisfy these constraints, then raise a ValueError exception.

All you are doing in this function is initializing variables, calling your recursive function, and returning the appropriate path. Don't write too much code--our solution has less than 10 lines! Test your code by uncommenting the code at the bottom of `ps5.py`.

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.