Lab 8: Graphs, Paths, Matrices

You are not logged in.

If you are a current student, please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.csail.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

This lab assumes you have Python 3.7 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: `lab8.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

This lab is worth a total of 4 points. Your score for the lab is based on:

- correctly answering the questions on this page (0.5 points)
- passing the test cases from `test.py` **and a set of server-only test cases** under the time limit (1.5 points), and
- a brief "checkoff" conversation with a staff member to share your code, including discussion of the bullets below and a review of your code's clarity/style (2 points).

For this lab, you will only receive credit for a test case if it runs to completion under the time limit on the server.

The questions on this page (including your code submission) are due at 4pm on Friday, Nov 15.

2) Introduction

In previous labs, we looked at algorithms on **unweighted graphs**, where all edges along a path contributed *equally* to the length of the path. In a **weighted** graph, edges can have *different contributions* to these path lengths. Edges with larger weights are considered "longer", while edges with smaller weights are considered "shorter."

In this lab, you will create two different implementations of an abstract, weighted graph interface (described in the next section in detail). The difference between the two implementations lies in the way they represent the graph itself. You have already seen one way to represent graphs in previous labs: the adjacency (or neighbor) dictionary, which can also be adapted for weighted graphs. In this lab we will also explore an alternative representation called the adjacency matrix.

Both representations have their own advantages, and depending on the the context in which they are used, one or the other might be a better choice in terms of performance (run-time) and space usage.

Both implementations of the `Graph` interface will provide support for retrieving shortest paths between *all* pairs of nodes. So far we have been computing shortest paths by searching the graph with breadth-first search (BFS), but this approach does not work with weighted graphs. We will therefore solve this *new* shortest path problem in two different ways, one for each `Graph` variant.

Note that the test cases work a bit differently in this lab. First, there will be both public and hidden test cases. Second, there are abstract and actual test classes. If you take a look at `test.py` and try to run each individual test class, you'll notice that for all the suites defined in `GraphTests` you'll get a TypeError. These cannot be run and their only purpose is to define what functionality each of the actual test classes will run. `Test_1_AdjacencyDictBasic`, `Test_2_AdjacencyMatrixBasic`, `Test_3_AdjacencyDictPaths`, `Test_4_AdjacencyMatrixPaths`, `Test_5_Factory`, `Test_6_CentralNodeDict` `Test_7_CentralNodeMatrix` are the "actual" test classes that you should try to run.

2.1) The `Graph` Interface

We provide the `Graph` interface at the top of `lab.py`. In Python, interfaces are called abstract base classes (ABCs), and their unimplemented (abstract) methods are marked with the `abstractmethod` decorator[1]. Any class that implements this interface must be a subclass of the interface. If any interface method remains unimplemented in this subclass, the Python interpreter will raise an error. You can find more details about this language feature here.

Since `Graph` is an interface, its methods may remain unimplemented. Indeed, any method which relies on a specific class's implementation should be implemented within that class, not within the interface. If a method need not rely on a class-specific implementation (that is, it can be implemented solely by using the interface's other methods), then implementing it in the interface is appropriate, to reduce code repetition. If you choose to implement one of the interface methods inside the interface, which you are encouraged to do if it can eliminate code repetition, then you should remove that method's `abstractmethod` decorator. During checkoff, you will be asked to justify where you implemented your methods.

You should not modify the "signatures" (name or required parameters) of the interface methods we have given to you, nor should you remove them, as we will rely on them to test your code.

Remember that the roles of an interface are to provide a consistent way of interacting with a data structure, and to hide the specifics of any particular implementation. This way, changes in the different implementations do not affect the methods that a user will use to interact with the graph.

Which of the following is not true?

--

This question is due on Friday November 15, 2019 at 04:00:00 PM.

The interface represents a mutable, directed, weighted graph, and contains the methods listed below. In our graphs, node names will be unique strings and edge weights will be nonnegative numbers.[2]

- `add_node(self, node)`: add a new node with name `node` to the graph. If a node with this name already exists, this method should raise a `ValueError`.
- `add_edge(self, start, end, weight)`: add a directed edge from node with name `start` to node with name `end`, with weight equal to `weight`, which is assumed to be a nonnegative real number. If either of these nodes doesn't exist, this method should raise a `LookupError`. If the edge already exists, then set its weight to the given value `weight`.
- `nodes(self)`: return the set of all node names in the graph.
- `neighbors(self, node)`: return the set of all tuples (`neighbor`, `weight`) for which `node` has an edge to `neighbor` with weight `weight`. If `node` doesn't exist, this method should raise a `LookupError`.
- `get_path_length(self, start, end)`: return the length of the shortest path from `start` to `end`. Return `None` if there is no such path. If either `start` or `end` doesn't exist, this method should raise a `LookupError`.

- `get_path(self, start, end)`: return the list of nodes (starting with `start` and ending with `end`) along a shortest path from `start` to `end`. Return `None` if there is no such path. If either `start` or `end` doesn't exist, this method should raise a `LookupError`.
- `get_all_path_lengths(self)`: return a dictionary mapping tuples `(u, v)` to real numbers which correspond to the length of the shortest path from `u` to `v`. If there is no path from `u` to `v`, `(u, v)` is not a key in the dictionary.
- `get_all_paths(self)`: return a dictionary mapping tuples `(u, v)` to lists of nodes (starting with `u` and ending with `v`) which correspond to the shortest path from `u` to `v`. If there is no path from `u` to `v`, `(u, v)` is not a key in the dictionary.

Throughout this lab, we use the convention that there is always a valid path from a node `'u'` to itself, the path `['u']`, which has length 0. (But `neighbors(self, node)` should not include `node`.)

3) Implementations of the Graph Interface

3.1) Adjacency Dictionary Graph

The first implementation of our interface will use an adjacency dictionary. You have already seen this in action in previous labs. An adjacency dictionary has nodes as *keys* and sets of the nodes' neighbors as *values*. Implement `AdjacencyDictGraph` as a subclass of `Graph`, using an adjacency dictionary as the internal representation.

Recall that, for this weighted graph, breadth-first search (BFS) cannot be used to compute shortest paths. Below, we will describe in detail a possible algorithm to achieve this, but for this version of `Graph` you are free to implement other weighted shortest path algorithms too if you wish.

> **Check Yourself:**
>
> Can you come up with an example of a weighted graph for which the shortest path that BFS would compute is not actually a shortest path based on the weights of the edges?

For our shortest path algorithm, we recommend augmenting your class with two new attributes, the `dist` and `pred` dictionaries. `dist` (distance) will map pairs of nodes to the distance between them. `pred` (predecessor) will map pairs of nodes `(u, v)` to the last node from which the node `v` was reached on the path from `u` to `v`. This last attribute is similar to the "parents" dictionary you may have used while implementing BFS in past labs. However, the predecessor can be different for node `v` when computing the shortest path from node `u` versus the shortest path from a different node `u'`.

First, initialize your `dist` dictionary to reflect the fact that there is a 0-length path from any node to itself, such that, before we run our algorithm, we assume that there are only infinite-length paths between distinct/non-equal nodes (i.e. there is no path between them). For this, you might find the `float('inf')` python syntax useful.

Next, implement a technique called edge relaxation, in which your dictionaries may be updated to reflect the discovery of a shorter path from `u` to `v`, through some intermediate node `x` which has an edge to `v`. Specifically, we recommend writing a helper method `relax` that takes three arguments `u`, `x`, and `v`. If the current distance (as stored in `dist`) between `u` and `v` is larger than the distance between `u` and `x` plus the weight of the edge from `x` to `v`, it should update the distance from `u` to `v` to be this smaller value. If this is the case, it should also update the predecessor of `v` along the path from `u` (as stored in `pred`) to be `x`. This information, that we arrived from `u` to `v` by passing through `x` last, will help us reconstruct the path from `u` to `v` later.

Now you should use edge relaxation to implement the core algorithm. In the algorithm, you need to perform edge relaxation for every node-edge pair `(u, (x, v))` in the graph, where `u`, `x`, and `v` could be different nodes (by our definition of neighbors, `v` and `x` will never actually be the same). You need to repeat this process $N - 1$ times, where N is the number of nodes in the graph.[3] After this, `dist` will store lengths of the shortest paths between all pairs of nodes, and `pred` will contain enough information to reconstruct the paths.

Finally, implement reconstruction of the paths. You will use the information stored in the `pred` dictionary. Complete the following questions to ensure you understand what is stored in `pred` after the completion of the algorithm.

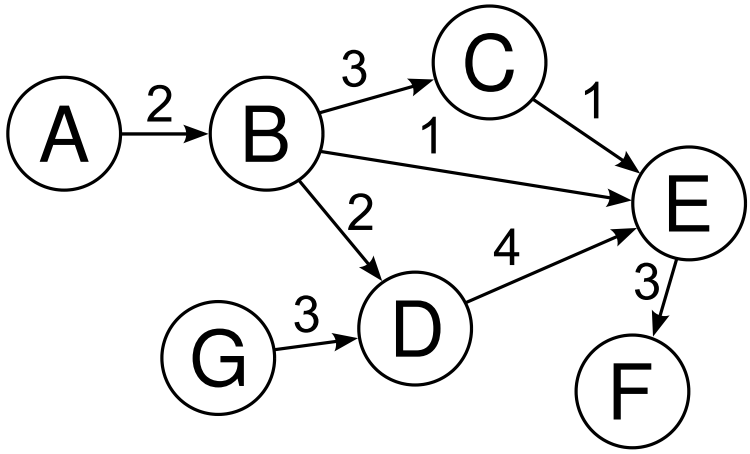

Figure 1. Weighted, directed graph.[4]

After running our algorithm on the graph above, what is stored in `pred` for the pair `'B'`, `'D'`? Your answer should be a string (recall that nodes are strings).

This question is due on Friday November 15, 2019 at 04:00:00 PM.

After running our algorithm on the graph above, what is stored in `pred` for the pair `'A'`, `'E'`? Your answer should be a string.

This question is due on Friday November 15, 2019 at 04:00:00 PM.

After running our algorithm on the graph above, what is stored in `pred` for the pair `'G'`, `'F'`? Your answer should be a string.

This question is due on Friday November 15, 2019 at 04:00:00 PM.

3.2) Adjacency Matrix Graph

Another way to represent graphs is using an adjacency matrix, which might be familiar from Recitation 6. This matrix is a 2D array of numbers, where the value at row `i` and column `j` is the weight of the edge from the node with ID `i` to the node with ID `j`. (Since node names are strings, but row and column numbers are integers, we associate node names with a particular row/column integer, which we call that node's ID.) If there is no edge from `i` to `j`, the value is infinity.

An adjacency matrix for the graph in Figure 1 is shown below.

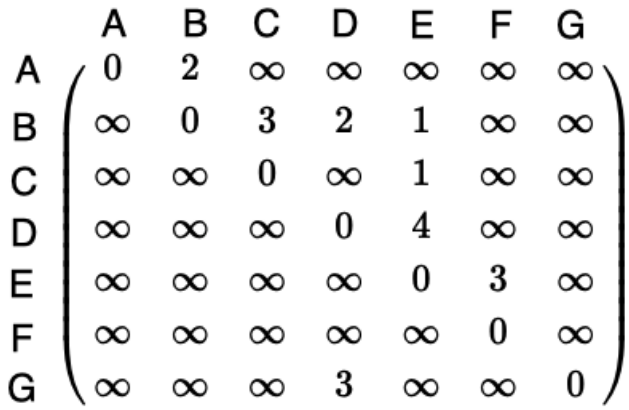

Figure 2. Adjacency matrix for the directed graph in Figure 1.

The advantage of this representation is twofold:

1. For "dense" graphs (where there are many more edges than nodes), it uses less space than the corresponding adjacency dictionary representation, because dictionaries usually use significantly more space than the amount just needed to store their keys.
2. It is equally easy to traverse edges `(u, v)` in their natural direction, from `u` to `v`, and in the opposite direction, from `v` to `u`, which can be an advantage for certain kinds of algorithms.

Implement `AdjacencyMatrixGraph` as a subclass of `Graph` using a matrix as the internal representation. Since this representation can only store edges between node IDs (indices), you will need a way to convert between node names (strings) and IDs. The `get_path`, `get_path_length`, `get_all_paths`, and `get_all_path_lengths` methods should return paths or path lengths computed via the algorithm described below, which is different from the algorithm used for `AdjacencyDictGraph`.

Check Yourself:

To get an intuition for the algorithm before seeing it in more detail, consider the original adjacency matrix. Recall that the value at row `i` and column `j` is the length of the path with just zero or one edge, from the node with ID `i` to the node with ID `j`. How would you use the values in that matrix to compute a new matrix in which the value at row `i` and column `j` is the length of the path with up to _two_ edges? First think about getting just one `i`, `j` entry of the new matrix.

Show/Hide

For this algorithm, you will need two additional data structures, not unlike the ones for the previous algorithm. Create a matrix `dist` of the same dimensions as the adjacency matrix, which will store the lengths of the shortest paths between nodes (as opposed to just the length of the edges directly between nodes, as is stored in the adjacency matrix). This roughly corresponds to the `dist` dictionary from the previous algorithm. Also create a `pred` matrix. This will serve a similar purpose as the `pred` dictionary used above. Here, the location at `(u, v)` in the matrix will have value equal to *some* node with index `x` on a shortest path from node at index `u` to node at index `v` (no longer necessarily the last one). There might be many other nodes between `u` and `x` and between `x` and `v`. This means we will need a slightly different way of reconstructing the paths, as we'll see later.

The fundamental building block of the algorithm is a procedure called `double_max_edges` that takes a matrix `M`, of the same shape as the adjacency and `dist` matrices, but instead contains the lengths of the shortest paths in the graph, only considering paths with at most l edges in them. The procedure returns a new matrix `M'` that contains the lengths of the shortest paths in the graph, now considering all possible paths with at most $2l$ edges in them. The procedure, which we recommend you implement as a helper function taking a single argument `M`, works as follows.[5]

First, initialize a new matrix `M'` to be a copy of `M`. Then, for every row `i` (representing a node `u`) and column `j` (representing another node `v`), go through all `k` possible values of each node `x`, and, if the distance (path length) from `u` to `v` in `M'` (stored as `M'[i][j]`) is larger than the sum of the original distance of `u` to `x` and `x` to `v` (stored as `M[u][x]` and `M[x][v]`,

respectively), then it updates the `u` to `v` distance in `M'` to be this sum. If it updates this distance, it also updates the predecessor for the `(u, v)` path in `pred` to be `x`.

The full algorithm should perform the `double_max_edges` procedure on the adjacency matrix repeatedly, until the maximum number of edges on a shortest path reaches $N - 1$, where N is the number of nodes in the graph. (This is the maximum possible number of edges on the shortest path between any two nodes. Convince yourself of why that is true.) In order to do this, you should keep track of this value, the maximum number of edges along a path in the matrix.

Finally, in order to reconstruct the shortest path between two nodes, you will need to think about what information `pred` is storing and how you can use it.

> **Check Yourself:**
>
> Let's say you have run a few rounds of the algorithm, and you have now found the path between `u` and `v` to be of length 10. However, in the next round, it turns out that there is a shorter path (that nevertheless traverses more edges) that goes through `x`, so the sum of the distance from `u` to `x` and `x` to `v` is less than 10. Where on the `u`–`v` path can `x` be? Can we say anything certain about how many edges away it is from either `u` or `v`?

As mentioned above, `pred[u][v]` stores ***some*** node along the shortest path from `u` to `v`. This means that if we want to reconstruct the full path from `u` to `v`, we have to recursively reconstruct the path from `u` to `x` and the path from `x` to `v`, then concatenate them.

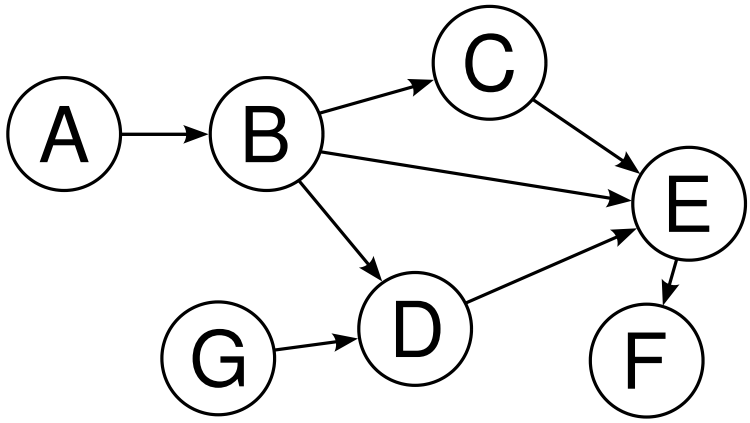

Figure 3. A directed graph with weights not shown.[6]

One way to build the `pred` matrix is to store `None` s in the cells (i, j) where either there is no path between the node with index i and the node with index j, or there is no other node between these two (i and j are connected).

The following concept questions refer to the graph from Figure 3 above with edge weights unknown to you and the following `pred` matrix (after running our algorithm). Also, when answering the concept question assume that in the `pred` matrix, index 0 refers to node A, index 1 refers to node B, index 2 to C, etc.

```
pred = [
        [   0, None,    1,    1,    1,    3, None],
        [None,    1, None, None,    3,    3, None],
        [None, None,    2, None, None,    4, None],
        [None, None, None,    3, None,    4, None],
        [None, None, None, None,    4, None, None],
        [None, None, None, None, None,    5, None],
        [None, None, None, None,    3,    3,    6]
        ]
```

What is the shortest path from node `"B"` to node `"D"`? Your answer should be a list of strings.

This question is due on Friday November 15, 2019 at 04:00:00 PM.

What is the shortest path from node `"A"` to node `"C"`? Your answer should be a list of strings.

This question is due on Friday November 15, 2019 at 04:00:00 PM.

What is the shortest path from node `"A"` to node `"F"`? Your answer should be a list of strings.

This question is due on Friday November 15, 2019 at 04:00:00 PM.

After implementing both versions of the `Graph` interface your code should pass all test cases in the `Test_1_AdjacencyDictBasic`, `Test_2_AdjacencyMatrixBasic`, `Test_3_AdjacencyDictPaths`, `Test_4_AdjacencyMatrixPaths` test classes.

Note that a different shortest paths algorithm for the matrix representation might not be efficient enough to pass all the test cases, so we recommend that you implement our approach.

4) Creating a Graph Factory

Now that we implemented both versions of the `Graph` interface, we will want to create instances of these classes. Of course, given a real graph, we could manually add all its nodes and edges to an empty `Graph` object (like `AdjacencyMatrixGraph()` or `AdjacencyDictGraph()`), but it would be much nicer to have another class do this for us!

The advantage of this approach is that this class (called a factory, since it produces objects) can choose between the different implementations based on certain properties of the graph. For example, due to the memory advantage that lists have over dictionaries in Python, a representation involving an adjacency matrix is more suitable for very dense graphs than the alternative representation, an adjacency dictionary.

With this in mind, we will build a factory class that, when given the nodes and edges to add to a graph, will decide the optimal implementation of the `Graph` interface to instantiate based on how dense or sparse the edges are.

Implement the class `GraphFactory` in `lab.py`. The `__init__` constructor takes a `cutoff` argument, which is a `float` (real number) between 0 and 1. This number is the maximum density[7] of the graph for which the factory should still instantiate an `AdjacencyDictGraph`. We define density to be the number of edges in the graph, divided by the maximum number of edges which *could* occur in a graph with the same number of nodes — for our directed graphs without self-loops (edges from nodes to themselves), this maximum number of edges is $N \times (N-1)$, where N is the number of nodes. More specifically, this instance of `GraphFactory` should create an `AdjacencyDictGraph` if the density of the graph to be created is less than or equal to `cutoff`. Otherwise, it should create an instance of `AdjacencyMatrixGraph`.

The only method in `GraphFactory` is `from_edges_and_nodes`, which takes in two lists as arguments, `weighted_edges` and `nodes`. `weighted_edges` represents the edges in the graph in the form of (`node1`, `node2`, `weight`) tuples, where the edge is going from `node1` to `node2`, with weight `weight`. `nodes` represents the node names as strings that uniquely identify the nodes to add to the graph. The `from_edges_and_nodes` method should return an instance of the Graph implementation that is optimal. **In this method, you should not rely on any methods other than those in the original `Graph` interface we gave you.**

To better understand how the `GraphFactory` class would be used, take a look at the following piece of code which shows the factory in practice:

```
>>> nodes = ["0", "1", "2", "3"]
>>> edges1 = [("0", "1", 3), ("2", "3", 0.5)]
>>> factory = GraphFactory(0.5)
>>> graph1 = factory.from_edges_and_nodes(edges1, nodes)
>>> type(graph1)

<class '__main__.AdjacencyDictGraph'>

>>> edges2 = [("0", "1", 2.1), ("0", "2", 8), ("0", "3", 2), ("1", "2", 1),
...           ("1", "3", 2), ("1", "0", 1.5), ("2", "0", 8), ("3", "0", 0),
...           ("2", "3", 4)]
>>> graph2 = factory.from_edges_and_nodes(edges2, nodes)
>>> type(graph2)

<class '__main__.AdjacencyMatrixGraph'>
```

In this example, we used a `cutoff` value of 0.5. This means that for any graph containing more than half of the maximum possible number of edges for its node count, the `from_edges_and_nodes` method picks `AdjacencyMatrixGraph`. For graphs with a number of edges that falls below or is equal to this cutoff, `AdjacencyDictGraph` will be chosen. Since the `edges1` list describes a sparse graph (with a density of $2/(4 \times 3) \leq 0.5$), its optimal representation would be an adjacency dictionary. In contrast, the `edges2` list describes a more dense graph, so an adjacency matrix is preferred.

Depending on where we use these graph implementations, we might need to choose more or less strict cutoffs to optimize for performance or space usage. The constructor (`__init__` function) of `GraphFactory` takes a `cutoff` parameter corresponding to this value, for this reason.

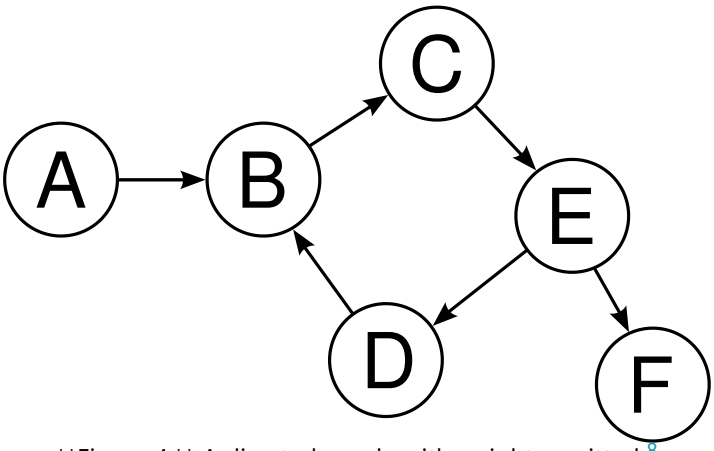

Figure 4. A directed graph, with weights omitted.[8]

Which implementation of `Graph` should a `GraphFactory` with `cutoff = 0.2` instantiate for the graph on Figure 4 above?

--

This question is due on Friday November 15, 2019 at 04:00:00 PM.

Which implementation of `Graph` should a `GraphFactory` with `cutoff = 0.1` instantiate for the graph on Figure 4 above?

--

This question is due on Friday November 15, 2019 at 04:00:00 PM.

At this point in the lab you should make sure that your code also passes all of the test cases inside the `Test_5_Factory` class. Note that there are also hidden test cases for this part, which you can run only by submitting your code to the website. These server-only tests will attempt to check that you have not used any implementation-specific methods or attributed in your `GraphFactory` implementation, and that you only used the functionality defined on the interface.

5) Using Your Graphs

As a final piece of this lab, you will put your path-constructing methods to use! You are interested in finding the node which is "most central" in a given graph. To determine how central a node is, you need to consider the average round-trip distance from that node to all other nodes. A round trip from `u` to `v` traverses the path from `u` to `v` and back from `v` to `u`.

Implement the function `get_most_central_node`. The function is given some instance of `Graph`. You should not rely on the specific implementation. The function should return the node in the graph with the shortest average round trip to any other node. That is, it returns a node for which round trips to all other nodes are possible, and which achieves the minimum possible average distance for those round trips. It is guaranteed that the graph has at least one node for which all round trips are possible.

Check Yourself:

What does this guarantee imply about an arbitrary node's ability to make all round trips? How could you use that to simplify your code for `get_most_central_node`?

As with the previous section, there are both public and hidden test cases for this part too. After adding support for computing the most central node, your code should pass all test cases, both public and server-only.

Finally, to see your code in action, create two representations of your favorite graph as an `AdjacencyDictGraph` and an `AdjacencyMatrixGraph` instance. Run `get_most_central_node` on both of them. What do you observe? How dense or sparse was your input graph? Which implementation was faster? **Prepare to discuss your findings during the check-off conversation.**

6) Code Submission

Scroll down to see the server-only test results! Those tests ensure that your `GraphFactory` and `get_most_central_node` do not rely on the specifics of your `Graph` implementations or interface methods added by you.

Select File No file selected

This question is due on Friday November 15, 2019 at 04:00:00 PM.

7) Checkoff

Once you are finished with the code, please come to a lab session or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code in detail before asking for a checkoff.** Since the clarity of your code will be evaluated as part of the checkoff, you may wish to take some time to comment your code, use good variable names, avoid repetitive code (create helper methods), etc.

Be prepared to discuss:

- Explanation of the trade-offs between using one `Graph` implementation or the other
- Your implementation of the shortest path algorithm for `AdjacencyDictGraph`
- Your implementation of the shortest path algorithm for `AdjacencyMatrixGraph`
- Justification of where you implemented methods (inside the interface or otherwise)
- Your code creating new graph instances inside `GraphFactory`
- Your implementation of `get_most_central_node`
- Demonstration of the performance of `get_most_central_node` on the two implementations with a graph of your choice

7.1) Grade

You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.

Footnotes

[1] For more information about decorators, see their entry in the official Python glossary.

[2] Our tests will also not contain zero-weight cycles.

[3] We need to repeat relaxing all edges this many times because, intuitively, for a given pair of nodes `u` and `v`, we can discover a node `x` with an updated path through which there exists a shortest path at most $N - 1$ times. Our algorithm is a variant of the Floyd-Warshall algorithm for finding shortest paths.

[4] Adapted from [Wikimedia Commons](https://commons.wikimedia.org/wiki/File:Directed_acyclic_graph.svg). Placed in the public domain.

[5] This procedure is also called `special_matrix_multiply` in some places, because the way a new matrix is computed is similar to multiplying a matrix by itself (i.e. squaring it). See this page for more information on that algorithm. Note, however, that the initialization of the matrix in the linked algorithm is different from that which we describe here.

[6] Source: [Wikimedia Commons](https://commons.wikimedia.org/wiki/File:Directed_acyclic_graph.svg). In the public domain.

[7] See this page for details on the density of graphs.

[8] Source: [Wikimedia Commons](https://commons.wikimedia.org/wiki/File:Directed_graph,_cyclic.svg). In the public domain.