Mini-project #1 – Python Programming

NAYA – Data Scientist Professional – Class of 2018

Background

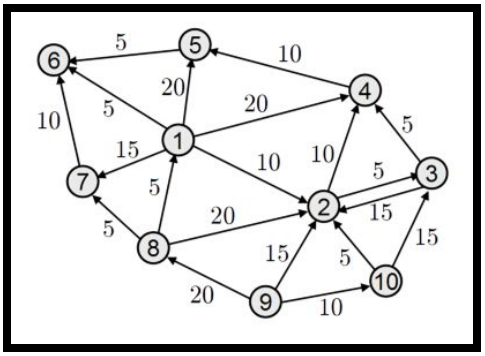
Terminology

This project is concerned with the implementation of mathematical graphs and their usages. The basic definition of a graph is a set of **nodes** where some pairs of nodes are connected by **edges**, and where these edges have **weights**. Each edge of the graph has a direction so by default a graph is **directional**. If there is an edge directed from node A to node B, then we say that A is **pointing at** B, and we note that this does not necessarily mean that B is pointing at B. However, if all the edges come in "pairs" (namely, if X is pointing at Y and Y is also pointing at X), then we say that the graph is **non-directional**.

If it is possible to "travel" from a node A to node B (either directly or through any number of consecutive neighbors), then we say that B is **reachable** from A and that there is a **path** connecting A to B. The weight of a path is the sum of the weights of the edges from which the path is made of.

Illustration

The figure on the right shows a graph with 10 nodes and 20 edges. Make sure you understand the following statements:



- The graph is directional, although it does include a single pair of nodes that are pointing at each other.
- Node "1" ("5" is the **name** of the node) is pointing at node "1" with an edge weight of 20, while "5" is not pointing at "5".
- "1" is reachable only from "8" and "9".
- "6" is reachable from "9" by several paths, and the minimal weight of such a path is 30.
- There is a path from "2" to "6", but there is no path from "6" to "2".

Submission

You should submit in one of the following formats:

1. a single py file
2. a single jupyter notebook

Grading

This project is not a test, it is a simulation of a code review process before submitting code for Git/ SVN.

Because the code is shared with other programmers, your code will be tested for readability, and clarity, please try to avoid complicated code because they are hard to fix and hard to debug.

The grading system will be based on 5 grades

1	The code is not running, have serious problems, it will require a talk and resubmission of the code
2	The code is working, some of the functionality is missing and will require another review before completing the review process and submitting the code to our simulated Git/SVN
3	The Code is running there are some changes that need to be made, but you can submit the code once you are done. I strongly advice to actually correct the remarks in order to make sure you understand them
4	The code is very good, It is well documented and clear to understand
5	A professional code, the author accounted for code architecture, efficiency, and readability.

Although the maximum score is 5 an extra point will be given for usage of design patterns (See GOF design patterns or more current design patterns) for a good implementation of the code, or for efficiently implementing a shortest path algorithm

Part I – The *Node* class

Task 1 – Define the class

Implement the *Node* class with the following properties:

- Attributes
  - *name* – the "name" of the node
    - *name* can be any immutable object, most naturally a string or a number.
  - *Pointing_at*– a dictionary of the edges with the nodes names as keys and the weights of the corresponding edges as values.
- Methods
  - *__init__(self, name)*
    - The method does <u>not</u> have to test the validity of *name*.
  - *__str__(self)*
  - *__len__(self)* – returns the number of neighbors
  - *__eq__(self, other)* – based on the *name* attribute
  - *__ne__(self, other)*
  - *is_pointing_at(self, name)* – returns True if *self* is pointing at *name*.
  - *add_neighbor(self, name, weight=1)* – adds *name* as a neighbor of *self*.
    - This method does <u>not</u> test whether a node named *name* exists.
    - This method should <u>not</u> allow adding a neighbor with a name of an existing neighbor.
    - This method should <u>not</u> allow adding a neighbor with the same name as *self*.
    - * Extra points, create pre validation test and print to the user the problems
  - *remove_neighbor(self, name)* – removes name from being a neighbor of self.
    - This method does <u>not</u> test whether a node named *name* exists.
    - This method should <u>not</u> fail if *name* is not a neighbor of *self*.
    - * Extra points, create pre validation test and print to the user the problems
  - *get_weight(self, name)* – returns the weight of the relevant edge.
    - This method should return *None* if *self* is not pointing at *name*.
  - *is_isolated(self)* – returns *True* if *self* has no nodes pointing at him

Task 2 – Exemplary usage

Question 1

Create 10 Node objects according to the figure above, print them (textually, of course).

Question 2

Make some tests to make sure your implementation works.

Question 3

How many edges are in the graph, and what is their total weight?

Question 4

Sort the nodes by the number of their neighbors.

Part II – The Graph class

Task 1 – Define the class

Implement the Graph class with the following properties:

- Attributes
  - *name* – the name of the graph.
  - *nodes* – this is a dictionary fully descriptive of the graph. Its keys are the names of the nodes, and its values are the node instances (of class Node).
- Methods
  - *__init__(self, name, nodes=[ ])*
    - *nodes* is an iterable of *Node* instances.
  - *__str__(self)*
    - This method should print the description of all the nodes in the graph.
    - Tip: the built-in function *print()* is not the only function that calls the *Node.__str__()* method, but also the built-in function *str()*.
  - *__len__(self)* – returns the number of nodes in the graph
  - *__contains__(self, key)* – returns *True* in two cases: (1) If *key* is a string, then if a node called *key* is in *self*, and (2) If *key* is a Node, then if a node with the same name is in *self*.
    - Tip: use the built-in function isinstance().
  - *__getitem__(self, name)* – returns the Node object whose name is *name*.
    - This method should raise KeyError if *name* is not in the graph.
  - *__add__(self, other)* – returns a new Graph object that includes all the nodes and edges of *self* and *other*
    - If a node exists both in *self* and in *other*, then the original node should not be updated.
    - Tip: Implement the method *add_node()* before you implement this method.
    - Validate that there is at least one common node
  - *add_node(self, node)* – adds a new node to the graph
    - Its input argument is a *Node* instance.
    - If a node with the same name already exists in the graph, then existing edges should not be overwritten, but new edges should be added.
    - * Extra points, create pre validation test and print to the user the problems
  - *remove_node(self, name)* – removes the node *name* from *self*.

        - This method should not fail if *name* is not in *self*.
        - * Extra points, create pre validation test and print to the user the problems
    - *is_edge(self, frm_name, to_name)* – returns *True* if *to_name* is a neighbor of *frm_name*.
        - This method should not fail if either *frm_name* is not in *self*.
        - * Extra points, create pre validation test and print to the user the problems
    - *add_edge(self, frm_name, to_name, weight=1)* – adds an edge making *to_name* a neighbor of *frm_name.*
        - This method should not fail if either *frm_name* or *to_name* are not in *self*.
        - If *to_name* is already a neighbor of *frm_name*, then the method should do nothing.
        - If *frm_name* and *to_name* are identical, then the method should do nothing.
        - * Extra points, create pre validation test and print to the user the problems
    - *remove_edge(self, frm_name, to_name)* – removes *to_name* from being a neighbor of *frm_name*.
        - This method should not fail if *frm_name* is not in *self*.
        - This method should not fail if *to_name* is not a neighbor of *frm_name*.
        - * Extra points, create pre validation test and print to the user the problems
    - *get_edge_weight(self, frm_name, to_name)* – returns the weight of the edge between *frm_name* and *to_name*.
        - This method should not fail if either *frm_name* or *to_name* are not in *self*.
        - This method should return *None* if *to_name* is not a neighbor of *frm_name*.
        - * Extra points, create pre validation test and print to the user the problems
    - *get_path_weight(self, path)* – returns the total weight of the given path, where path is an iterable of nodes' names.
        - This method should return *None* if the path is not feasible in *self*.
        - This method should return *None* if *path* is an empty iterable.
        - Tip: The built-in functions *any()* and *all()* regard nonzero numbers as *True* and *None* as *False*.
    - *is_reachable(self, frm_name, to_name)* – returns *True* if *to_name* is reachable from *frm_name*.
        - This method should not fail if either *frm_name* or *to_name* are not in *self*.
    - *find_shortest_path(self, frm_name, to_name)* – returns the path from *frm_name* to *to_name* which has the minimum total weight.

- This method should return *None* if there is no path between *frm_name* and *to_name*.
- Note: path finding is usually implemented with recursion. We didn't learn recursion in our course, so I recommend implementing a non-recursive algorithm like "breadth-first search" or "depth-first search".

Task 2 – Exemplary usage

Question 1

Create 3 Graph objects, each contains a different collection of nodes, which together contain all 10 nodes. Use the __add()__ method to create a total graph that contains the entire data of the example.

Question 2

Make some tests to make sure your implementation works.

Question 3

Sort the nodes by the number of their reachable nodes.

Question 4

What is the pair of nodes that the shortest path between them has the highest weight?

Task 3 – The roadmap implementation

The files travelsEW.csv and travelsWE.csv record a large number of travels made by people from five regions in the country, called Center, North, South, East and West.

Question 1

From each file create a graph whose nodes are the country regions, and whose edges are the roads themselves (if a travel was not recorded between country regions, then it means such road does not exist). The weight of each edge is defined as the average time (in seconds) of all the travels done on that road. When the two graphs are ready, add them together to create the complete graph of the roadmap.

Question 2

From which region to which region it takes the longest time to travel?

*Bonus part

Part III – Non-directional graph

Task 1 – define the class

Implement the *NonDirectionalGraph* class as a sub-class of Graph. The main property of the non-directional graph is that its edges come in pairs, so if an edge is added or remove, then the class must make sure that the same applies to its counterpart. There are no additional methods to implement, but make sure to overwrite the relevant methods.

Task 2 – The social network implementation

The file social.txt describes chronologically the intrigues among 14 friends. Use the data in the file and the classes you've defined to answer the following questions.

Question 1

What was the highest number of simultaneous friendships?

Question 2

What was the maximum number of friends Reuben had simultaneously?

Question 3

At the current graph (considering all the data of the file), what is the maximal path between nodes in the graph?

Question 4

Implement a function called *suggest_friend(graph, node_name)* that returns the name of the node with the highest number of common friends with *node_name*, which is not already one of his friends.

Good luck!