Assignment Overview

You will be implementing a graph data structure. There are four user-defined classes that you will be working with: Graph, Vertex, Edge, and Path.

- The Graph class represents a Directed Acyclic Graph (DAG). You will be using an adjacency map like the one described in the textbook as the underlying data structure.
- Vertex represents a single vertex in the graph. Each vertex will have a unique id associated with it.
- Edge represents a directed edge from a source vertex to a destination vertex.
- Path objects contain a list of vertex ids in the order that they were visited, and the total weight of the path.

Assignment Deliverables

Be sure to use the specified file name(s) and submit them for grading **via D2L Dropbox** before the project deadline.

- Graph.py

Assignment Specifications

Your task will be to complete the methods listed below.

For Vertex:

- **add_edge(destination, weight):**
  - Creates an edge object and adds to the list of edges.
- **degree:**
  - Returns the number of outgoing edges that vertex has.

- **get_edge(destination):**
    - Returns the *Edge* that goes to a specified destination node. If the edge is not found, return None
- **get_edges():**
    - Returns a list of all the *Edge* objects this vertex has.

For Graph:

- **construct_graph:**
    - Adds all edges created in generate_edges to the graph. generate_edges will return a list of lists containing edges in this format: [source, destination, weight]
    - Uses the dictionary *self.adj_map* to store vertices' IDs as keys and their objects as values.
    - Return(None)
- **vertex_count:**
    - Returns the number of vertices in the graph.
    - Method should run in O(1) time
- **vertices:**
    - Returns a list of all *Vertex* objects in the graph.
    - Method should run in O(V) time
- **insert_edge(source, destination, weight) :**
    - Inserts a new *Edge* from source to destination with a specified weight. If the edge is already in the graph, **replace** the weight.
    - Method should run in O(source's degree) time
    - Return(None)
- **find_valid_paths(source, destination, limit):**
    - Finds all valid paths between two vertices in the graph. A path is valid if the total accrued weight along the path **does not** exceed the limit. Store those valid paths as Path objects.
    - Path objects contain an ordered list of visited vertices and the total weight along the path
    - Return(python list[Path])
    - Worst case time complexity: O((V-1)!)
- **find_shortest_path(source, destination, limit):**
    - Return a valid *Path* with the smallest total weight. If there are multiple paths, return any one.
    - Worst case time complexity: O((V-1)!)
    - Return(Path)
- **find_longest_path(source, destination, limit):**

    - Return a valid *Path* with the largest total weight. If there are multiple paths, return any one.
    - Worst case time complexity: O((V-1)!)
    - Return(Path)
- **find_most_vertices_path(source, destination, limit):**
    - Return a valid *Path* that visits the most vertices. If there are multiple paths, return any one.
    - Worst case time complexity: O((V-1)!)
    - Return(Path)
- **find_least_vertices_path(source, destination, limit):**
    - Return a valid *Path* that visits the least vertices. If there are multiple paths, return any one.
    - Worst case time complexity: O((V-1)!)
    - Return(Path)

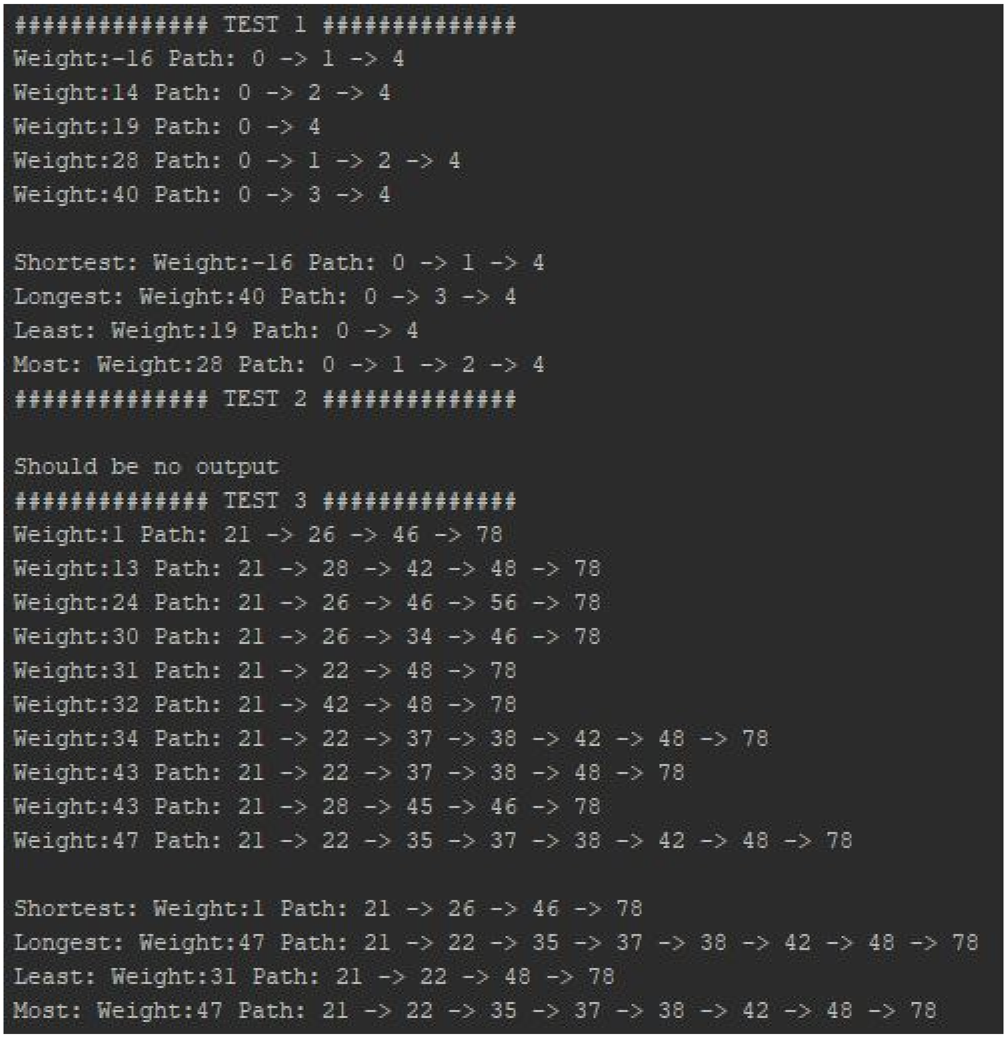
You can make additional helper functions, if useful.

Points will be deducted if your solution has any warnings of type:

- Path and Edge objects are fully implemented and no part of the class definitions should be modified.
- The newest distribution python 3.6 interpreter will be used to execute your solution.
- Any method or class that is marked “do not edit” should not be altered
- You are required to complete the docstrings for any unmade and created function signatures.
- To test your classes, main.py is provided. Compare your results to the output below. It is recommended you also create test cases yourself to test for various edge cases.
- **Note:** your output might not match the screenshot for the shortest, longest, most, and least paths
- Errors when using your solution that cause the grading script to fail will result in a 25% deduction.
- You may not change any function signatures in anyway, which include class definitions.
- Your solution will be graded and tested against the equivalent equality operators and **not** standard output.

Testing your work

<u>Run your project on Pycharm, see sample run below of **main.py**</u>



```
############## TEST 1 ##############
Weight:-16 Path: 0 -> 1 -> 4
Weight:14 Path: 0 -> 2 -> 4
Weight:19 Path: 0 -> 4
Weight:28 Path: 0 -> 1 -> 2 -> 4
Weight:40 Path: 0 -> 3 -> 4

Shortest: Weight:-16 Path: 0 -> 1 -> 4
Longest: Weight:40 Path: 0 -> 3 -> 4
Least: Weight:19 Path: 0 -> 4
Most: Weight:28 Path: 0 -> 1 -> 2 -> 4
############## TEST 2 ##############

Should be no output
############## TEST 3 ##############
Weight:1 Path: 21 -> 26 -> 46 -> 78
Weight:13 Path: 21 -> 28 -> 42 -> 48 -> 78
Weight:24 Path: 21 -> 26 -> 46 -> 56 -> 78
Weight:30 Path: 21 -> 26 -> 34 -> 46 -> 78
Weight:31 Path: 21 -> 22 -> 48 -> 78
Weight:32 Path: 21 -> 42 -> 48 -> 78
Weight:34 Path: 21 -> 22 -> 37 -> 38 -> 42 -> 48 -> 78
Weight:43 Path: 21 -> 22 -> 37 -> 38 -> 48 -> 78
Weight:43 Path: 21 -> 28 -> 45 -> 46 -> 78
Weight:47 Path: 21 -> 22 -> 35 -> 37 -> 38 -> 42 -> 48 -> 78

Shortest: Weight:1 Path: 21 -> 26 -> 46 -> 78
Longest: Weight:47 Path: 21 -> 22 -> 35 -> 37 -> 38 -> 42 -> 48 -> 78
Least: Weight:31 Path: 21 -> 22 -> 48 -> 78
Most: Weight:47 Path: 21 -> 22 -> 35 -> 37 -> 38 -> 42 -> 48 -> 78
```