Weighted Graphs

Problem

Weighted graphs show up as a way to represent information in many applications, such as communication networks, water, power and energy systems, mazes, games and any problem where there is a measurable relationship between two or more things. It is therefore important to know how to represent graphs, and to understand important operations and algorithms associated with graphs. For this project, you will implement a directed, weighted graph and associated operations along with breadth-first search and Dijkstra's Shortest Path algorithms.

There are a number of nice Python modules for representing, displaying and operating on graphs. You are not allowed to use any of them for this project.

For this project, you will write a Graph ADT and a small main function as a small test driver "application". Include main() in your graph.py source file with conditional execution. It is common for modules to include a runnable main() to use for testing purposes. It happens in this case, you will have both main() AND the test code we give you to test your implementation.

Graph ADT

Your Graph ADT will support the following operations:

add_vertex(label): add a vertex with the specified label. Return the graph. label must be a string or raise ValueError

add_edge(src, dest, w): add an edge from vertex *src* to vertex *dest* with weight *w*. Return the graph. validate src, dest, and w: raise ValueError if not valid.

float get_weight(src, dest) : Return the weight on edge *src-dest* (math.inf if no path exists, raise ValueError if src or dest not added to graph).

dfs(starting_vertex): Return a generator for traversing the graph in depth-first order starting from the specified vertex. Raise a ValueError if the vertex does not exist.

bfs(starting_vertex): Return a generator for traversing the graph in breadth-first order starting from the specified vertex. Raise a ValueError if the vertex does not exist.

list dsp(src, dest): Return a tuple (path length , the list of vertices on the path from dest back to src). If no path exists, return the tuple (math.inf, empty list.)

dict dsp_all(src): Return a dictionary of the shortest weighted path between *src* and all other vertices using Dijkstra's Shortest Path algorithm. In the dictionary, the key is the the destination vertex label, the value is a list of vertices on the path from src to dest inclusive.

__str__: Produce a string representation of the graph that can be used with print(). The format of the graph should be in GraphViz dot notation, which is explained at https://graphs.grevian.org/example and many other places on the web. See Figure 1.

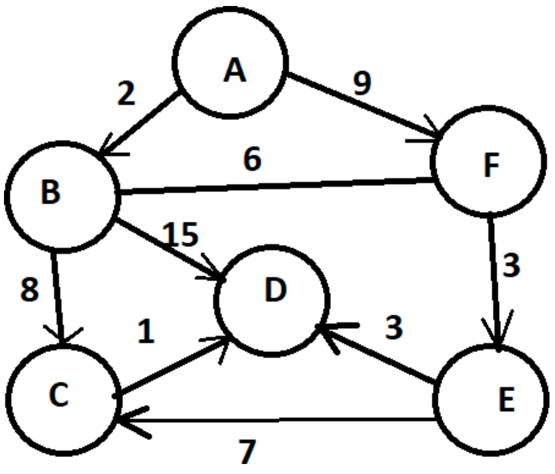
Python Generators

A good explanation of a python generator is found at
https://wiki.python.org/moin/Generators

Displaying Output of Graph Operations on an Example Graph G

In main(), you will do the following:

1. Construct the graph shown in Figure 1 using your ADT.
2. Print it to the console in GraphViz notation as shown in Figure 1.
3. Print results of DFS starting with vertex “A” as shown in Figure 2.
4. BFS starting with vertex “A” as shown in Figure 3.
5. Print the path from vertex “A” to vertex “F” (not shown here) using Djikstra’s shortest path algorithm (DSP) as a string like #3 and #4.
6. Print the shortest paths from “A” to each other vertex, one path per line using DSP.



The output of print(G) might look like Figure 1. Exact order does not matter as long as the edges are correct.

```
digraph G {
    A -> B[label="2.0",weight="2.0"];
    A -> F[label="9.0",weight="9.0"];
    B -> C[label="8.0",weight="8.0"];
    B -> D[label="15.0",weight="15.0"];
    B -> F[label="6.0",weight="6.0"];
    C -> D[label="1.0",weight="1.0"];
    E -> C[label="7.0",weight="7.0"];
    E -> D[label="3.0",weight="3.0"];
    F -> B[label="6.0",weight="6.0"];
    F -> E[label="3.0",weight="3.0"];
}
```

Figure 1. Directed graph G printed to console using GraphViz dot notation.

If this code were run:

```
    print("starting BFS with vertex
    A") for vertex in G.bfs("A"):
        print(vertex, end = "")
   print()
```

the output would look like:

```
 Starting BFS with Vertex A
 ABFDCE
```

Figure 2. Example of Breadth-First Traversal on example graph G.

If this code were run:

```
print("starting DFS with vertex
A") for vertex in G.dfs("A"):
    print(vertex, end =
"") print()
```

the output would look like:

```
Starting DFS with Vertex A
AFEDCB
```

Figure 3. Printing the output of Depth-First Traversal on example graph G.

What to Submit

Submit in Canvas as `project7.zip`:

1. `graph.py` → contains your graph ADT and all the operations and conditional main().

Grading (100 points)

pylint will be run on graph.py. Expected minimum score is 8.5.

Score is the sum of:

- Percentage test cases passed x 80 points
- min(Coding style score/8.5, 1) x 20 points
- Possible adjustment due to physical inspection of the code, to make sure you actually implemented things.