Weighted Graphs

Problem

Duration: 8 hours

Weighted graphs are used to model 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 several nice Python modules for representing, displaying and operating on graphs. *You are **not** allowed to use any of them 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. You will have both main() AND the test code we give you to test your implementation. See below for suggestions on what main should do.

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.

vertices(): Return a list of vertices in sorted in natural ascending order.

edges(): Return a list of edges. Each edge is a triple (src,dest, weight). The list of edges is not ordered.

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 list of vertices in depth-first traversal order from the starting vertex. Siblings should be visited in natural order. Raise a ValueError if the vertex does not exist.

bfs(starting_vertex): Return a list of vertices in breadth-first traversal order from the starting vertex. Siblings should be visited in natural order. Raise a ValueError if the vertex does not exist.

list dsp(src, dest): Return list with two items: the path length as a number and the list of vertices on the path from dest back to src as a list. If no path exists, return the list [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 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 2.

Suggested main()

The following are suggested actions for your main():

1. Construct the graph shown in Figure 1 using your ADT.
   (Alternative) If your instructor requires you to read an arbitrary graph from an input file, then your input file should use GraphViz dot notation like that shown below.
2. Print it to the console in GraphViz notation like that shown in Figure 2.
3. Print results of DFS starting with vertex "A" as demonstrated in Figure 3.
4. BFS starting with vertex "A" as shown in Figure 4.
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. If you have a different graph, show the same operation.
6. Print the shortest paths from "A" to each other vertex, one path per line using DSP.

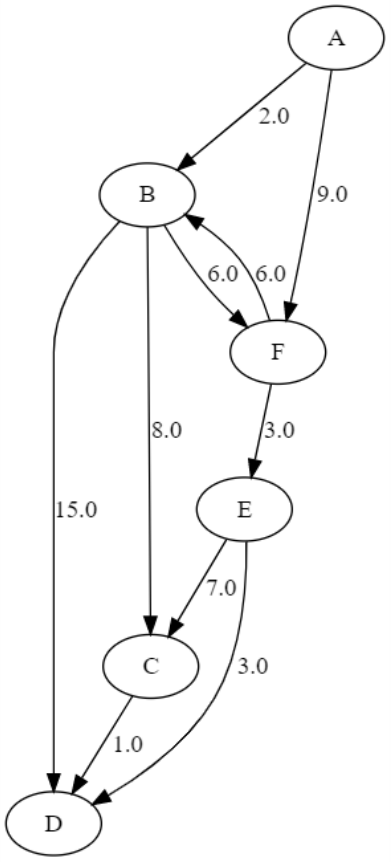

Figure 1. Graph drawn using GraphViz dot layout.

The output of print(G) might look like Figure 2. Exact order does not matter as long as the edges are correct. Note that there are newlines after the opening curly brace and the closing curly brace as well as after each semicolon. Each line between the curly braces is indented with 3 spaces. Proper formatting is required to pass the corresponding test case.

```
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 2. 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 3. 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 4. 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 implementation and all the operations and conditional main().
2. (optional, check with instructor) A link to a short video showing results of your code running, about 1 minute max.
   Post the video on Youtube, for example, with a unlisted link, and submit the link in comments to your submission.

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 implemented things.