

Weighted Graphs

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. Write your own.

Graph ADT (`graph.py`)

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 dijkstra_shortest_path(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 dijkstra_shortest_path(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 vertex label, the value is a tuple (path length , the list of vertices on the path from key back to *src*).
- `__str__`: Produce a string representation of the graph that can be used with `print()`.

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

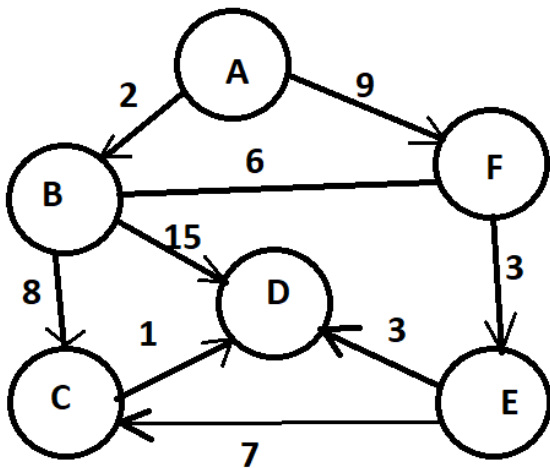
A good explanation of a Breadth First Search (or Traversal) is found at
https://en.wikipedia.org/wiki/Breadth-first_search

A good explanation of a Depth First Search (or Traversal) is found at
https://en.wikipedia.org/wiki/Depth-first_search

A good explanation of Dijkstra's algorithm (including pseudo code) is found at
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Displaying Output of Graph Operations on an Example Graph G

Suppose we create a graph G.



The output of `print(G)` might look like Figure 1:

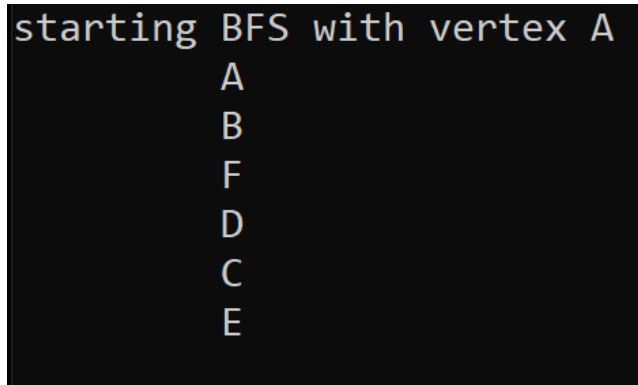
```
numVertices: 6
Vertex    Adjacency List
A         [('B', 2.0), ('F', 9.0)]
B         [('F', 6.0), ('D', 15.0), ('C', 8.0)]
C         [('D', 1.0)]
D         []
E         [('C', 7.0), ('D', 3.0)]
F         [('E', 3.0)]
```

Figure 1. Example of graph printed to console.

If this code were run:

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

the output would look like:

A terminal window with a black background and white text. The first line is "starting BFS with vertex A". The subsequent lines are indented with a tab character and list the vertices A, B, F, D, C, and E in that order from top to bottom.

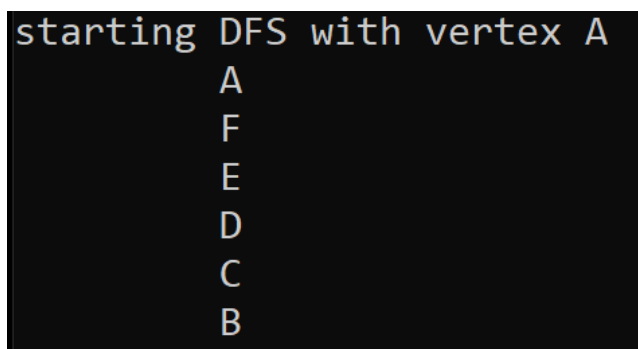
```
starting BFS with vertex A
\t A
\t B
\t F
\t D
\t C
\t E
```

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("\t", vertex)
```

the output would look like:

A terminal window with a black background and white text. The first line is "starting DFS with vertex A". The subsequent lines are indented with a tab character and list the vertices A, F, E, D, C, and B in that order from top to bottom.

```
starting DFS with vertex A
\t A
\t F
\t E
\t D
\t C
\t B
```

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

Unit Testing

Test Graph Creation

- create graph
- add vertex, assert ValueError
- add vertex "A", add vertex "B", assert returned value is instance of Graph
- add edge from "A" to "cat", assert ValueError
- add edge from "A" to "B" of weight "cat", assert ValueError
- add edge from "A" to "B" of weight 10.0, assert returned value is instance of Graph
- assert weight from "A" to "B" is 10.0
- assert weight from "B" to "A" is math.inf

TestGraph Traversals

- create a graph with vertices "A" – "F"
- add some edges
- assert that a generator is returned from bfs()
- assert that data created from generator is correct
- assert that a generator is returned from dfs()
- assert that data created from generator is correct

Test __str__

- create a graph (G) with vertices and edges
- call str(G) and validate the returned string

Test Shortest Paths

- create a graph with size vertices and nine edges
- assert dijkstra_shortest_path() from "A" to every other vertex (path will exit)
- assert dijkstra_shortest_path() from "D" to 'A' has distance of math.inf and an empty path list
- assert dijkstra_shortest_path() from "A" returns the proper dictionary
- assert dijkstra_shortest_path() from "D" returns the proper dictionary

Test Code Quality

- assert pylint on graph.py is 8.5 or higher

Grading (100 points)

- test graph creation 5
- test __str__ 5
- test Traversals 20
- test Shortest paths 60
- test Code Quality 10

Files to turn in through Canvas

- graph.py