Project 6: Weighted Graphs

Due: Friday April 12, 11:59 pm

1 Assignment Overview

For this project you will be implementing a weighted graph. A weighted graph, $G = (V, E)$, is defined as a set of *vertices*, $V = \{v_0, v_1, \ldots, v_n\}$, and a set of weighted *edges*, $E = \{(u_0, v_0, w_0), (u_1, v_1, w_1), \ldots, (u_m, v_m, w_m)\}$. An edge connects two vertices, u and v, and specifies a weight on that connection, w. Note that edges are undirected and, therefore, symmetric; (u, v, w) means the same thing as (v, u, w).

Your graph will have pathfinding capabilities; It is able to find the minimum weight path between two vertices.

Your graph will also be able to report if it is *bipartite* using a graph coloring algorithm.

2 Assignment Deliverables

You must turn in completed versions of the following files:

- Graph.py

Be sure to use the specified file name and to submit your files for grading via **Mimir** before the project deadline.

3 Assignment Specifications

Your task will be to complete the methods listed below:

- `insert_edge`: Adds an edge of the given weight between the two vertices.
- `degree`: The number of edges that intersect the given vertex.
- `are_connected`: Checks whether an edge exists between two vertices.
- `is_path_valid`: Determines whether an edge exists along each step of the path.
- `edge_weight`: Finds the weight of an edge between two vertices.
- `path_weight`: Finds the total weight of a path (sum of edge weights).
- `does_path_exist`: Checks whether a path exists between two vertices.
- `find_min_weight_path`: Finds a path of minimum weight between two vertices.
- `is_bipartite`: Determines if the graph is bipartite.

You may use any of the collections classes included in the default Python distribution that you need to store your graph. Your graph will have two member variables: `order`, which is the number of vertices, and `size`, which is the number of edges. Your `insert_edge` method should update the `size` member. You may add other member variables in your constructor. The memory requirements for your graph must be $O(|V| + |E|)$. Only the constructor and the `insert_edge` method may modify your graph. You will lose points if any of the other methods modify your graph.

Your `insert_edge`, `degree`, `are_connected`, and `edge_weight` functions should all run in constant time. `is_path_valid` and `path_weight` should run in time linear in the length of the input list. Each of them should also use a fixed amount of extra memory. `does_path_exist` and `is_bipartite` should use at most $O(|V|+|E|)$ time and $(|V|)$ extra memory. The `find_min_weight_path` method should use $O(|E|+|V|\log|V|)$ time and $O(|V| + |E|)$ extra memory.

The `is_bipartite` method will determine if the graph is a bipartite graph. In a bipartite graph, the vertices can be separated into two sets; all edges in the graph will connect vertices in different sets. This can be done using a coloring strategy; each vertex is colored either red or blue, the opposite of the color of one of its neighbors. If two adjacent vertices have the same color then the graph is not bipartite.

Each of your methods should raise an `IndexError` if one of the vertices is not in the graph. You may assume that the paths input into `is_path_valid` and `path_weight` are non-empty (they contain at least one vertex). You may not assume that all of the edges in the path exist. Both `edge_weight` and `path_weight` must return infinity if an edge is missing. Your `find_min_weight_path` should raise a `ValueError` if no path exists. No other exceptions should be raised by your program. `insert_edge` inserting an edge that already exists should update the weight on that edge.

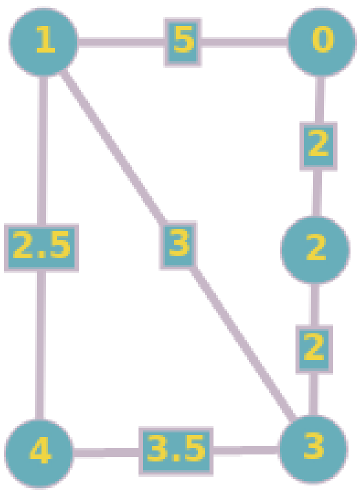
You should include comments where appropriate. In particular, you should describe the method used for storing edges (edge list, adjacency matrix, adjacency list, etc.). You must also add documentation for each of the above methods (as well as for any helper functions created) in the same style as for the pre-defined methods.

4 Assignment Notes

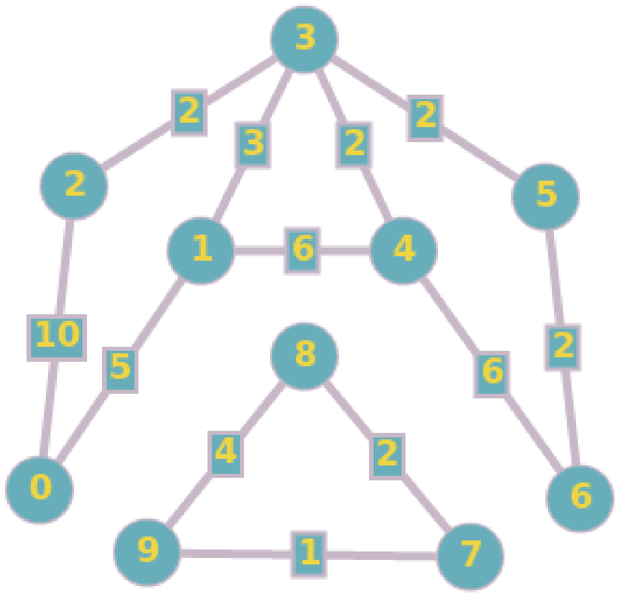
- Points will be deducted if your solution has warnings of any type.
- You have been supplied with stubs for the required methods. You must complete these methods to complete the project. You may add more functions than what was provided, but **you may not modify the signatures of the provided methods**.
- You do not have to worry about error checking for valid input. You may assume that the supplied reader function correctly reads the input file.
- You have been provided with some unit tests that can be used to assess your solution. There are additional test cases that will be kept hidden.
- It is your responsibility to check for potential edge cases. The supplied unit tests do not account for all possible valid inputs
- Several sample graphs have been provided. Visualizations for two of them are shown in Figures 1b and 1c above.
- Note that the graphs may not be connected. For example, the provided g2 graph is not connected but consists of two components. You must be able to determine if a disconnected graph is bipartite.
- You may use a wide variety of collections classes. You may want to use `list`, `set`, `dict`, `heapq`, `deque`, or `defaultdict`.

```
5
0  1  5.0
0  2  2.0
2  3  2.0
1  3  3.0
4  1  2.5
4  3  3.5
```

(a) g1.txt



(b) Visual representation of g1



(c) Visual representation of g2

Figure 1: Sample Input