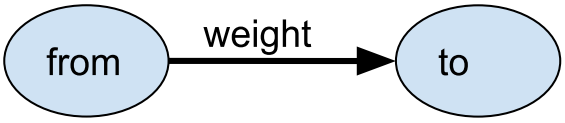
CS146: Spring 2023
Homework 6: Graphs
Due **Friday**, May 5, at 11:59PM
100 points

For this homework you will implement the Graph Interface which contains four methods on graphs. Graphs can be directed or undirected with either weighted edges or edges with equal weight (equivalent to an unweighted graph). **To get started, import the starter files: Vertex.java, Edge.java, Color.java, and Graph.java into the graph package you create in a new Java Project.** You **CANNOT modify** any of the given java files. Note that this homework does not require you to create JUnit tests. However just because I am not requiring you to create tests does not mean that you should not test your code. Testing your code frequently throughout the development process is an integral part of being a successful computer scientist (and it makes the debugging process significantly easier).

`Vertex.java`

This class represents a vertex in the graph. Note that the Vertex class contains several instance variables that you can use (or repurpose) to implement the graph algorithms. The equals/compareTo/hashCode only depend on the value of the vertex. You can assume that any graph has unique vertex values.

`Edge.java`

This class represents an edge in the graph. This class contains a `from` Vertex, a `to` Vertex, and a `weight`. This is equivalent to 

.

`Color.java`

This is an enum that represents different node colors that can be used when implementing depth first search.

`Graph.java`

This is an interface that requires you to implement 4 methods described below. Note that each method returns an ArrayList<Integers>. These are the values of the vertices.

Implementation

You must implement the class name and method signatures exactly as described below.

GraphImp.java (10 points)

You will implement the Graph interface in a class called GraphImp.

- It must contain the following instance variable that is package accessible (i.e., not private).
  - `Map<Vertex, ArrayList<Edge>> graph;`
- The constructor must have the following signature
  - `public GraphImp(Collection<Vertex> vertices, Collection<Edge> edges)`
  - The graph should list vertices in numerical order and each arraylist of edges should be ordered in increasing order by the value of the to vertex. You should think about the proper Collections to use to store the graph, vertices, and edges. You will justify your choices during the grading interview.
- The getter method that returns the vertices in the graph
  - `public Collection<Vertex> getVertices();`

Depth First Search (30 points)

You will implement the depth first search algorithm as described in the textbook (i.e., you will color vertices and keep track of the parent vertex in the DFS forest) except you will use a Stack instead of recursion. To get credit for this method you must use only a single stack and no recursive calls. You should return the ArrayList that represents the values of the vertices encountered on the depth first traversal. The traversal should begin at the vertex with lowest value. After completing this method. I should be able to look up the discovery times, finish times, and parent vertex for each vertex in the graph.

Topological Sort (40 points)

You will implement two versions of topological sort. One that uses a depth first search strategy and one that uses a queue. Both algorithms were described in class and are on the annotated slides.

1. `public ArrayList<Integer> topologicalSortDFS()`
   This method should return the topological sort using a depth first search.
   `topologicalSort(G)`
   ```
   1) Call DFS to compute the finishing times for each vertex
   2) As each vertex is finished put it at the front of a linked list
   3) Return the linked list of vertices
   ```

2. `public ArrayList<Integer> topologicalSortQueue()`
   This method should return the topological sort using a queue based on the indegree of the vertices.

```
topologicalSort(G)
Create a queue, Q, of vertices
Calculate the in degrees of each vertex
Add vertices with in degrees of 0 to the queue
while Q ≠ Ø
   v = Deque(Q)
   add v to my topological sort
   decrement the in degree of each vertex adjacent to v
   add vertices to Q if they now have in degree 0
```

In a comment above each method, answer the following question. *Can this method detect if a cycle is present in the graph? Why or why not?*

Dijkstra's Shortest Path (20 points)

You will implement Dijkstra's shortest path algorithm to find the shortest path between two vertices in the graph. You can assume that both vertices are in the graph and that the target vertex is reachable from the source vertex. We will discuss Dijkstra's algorithm on Tuesday, 5/2.

`public ArrayList<Integer> shortestPath(int source, int target)`
You should return the sequence of values that represent the shortest path from the source to the target starting with the source and ending with the target. You do not need to use a priority queue to implement Dijkstra's shortest path algorithm. The priority queue in Java does not support the decrease key operation. You can use whatever data structure/methods you want to figure out what vertex to add next. It doesn't need to be efficient.

Submitting

Export your project and upload it to Canvas. You can export your project as either a jar or a zip file. Make sure that the file you submit contains all your .java files in the package. If you upload a file that is missing any of the java source files you will get an automatic deduction of 10 points.