CSDS 233 Assignment #6

Due December 6, 2024, before 11:59 pm. 100 points

Submission Instructions

- The submissions will be evaluated on completeness, correctness, and clarity.
- Please provide sufficient comments in your source code to help the TAs read it.
- Please generate a single zip file containing all your *.java files needed for this assignment (not .class) and optionally a README.txt file with an explanation about added classes and extra changes you may have done.
- Name your file P6_YourCaseID_YourLastName.zip for your **coding exercises**. Submit your zip file electronically to Canvas.
- Please submit a **PDF** for your answers to the written exercise **separate from the zip file**.

Office Hours for This Assignment

For Written Exercises: 10:00 am - 11:00 am, December 4, 2024, via zoom.
Here is the zoom link:
https://cwru.zoom.us/j/2336305061?pwd=yN4eRRA0xbpFdnBy7kqxNxQ5oklo45.1
Passcode: 313723
For Programming Exercises: 16:00 pm - 17:00 pm, November 28, 2024, via zoom.
Here is the zoom link:
https://cwru.zoom.us/j/8153265328?pwd=S0JUSWVyQWF2aEZ4MDY4UnM3L3E4QT09
Passcode: 666661

Contact Information

If you have questions outside of the scheduled office hours:

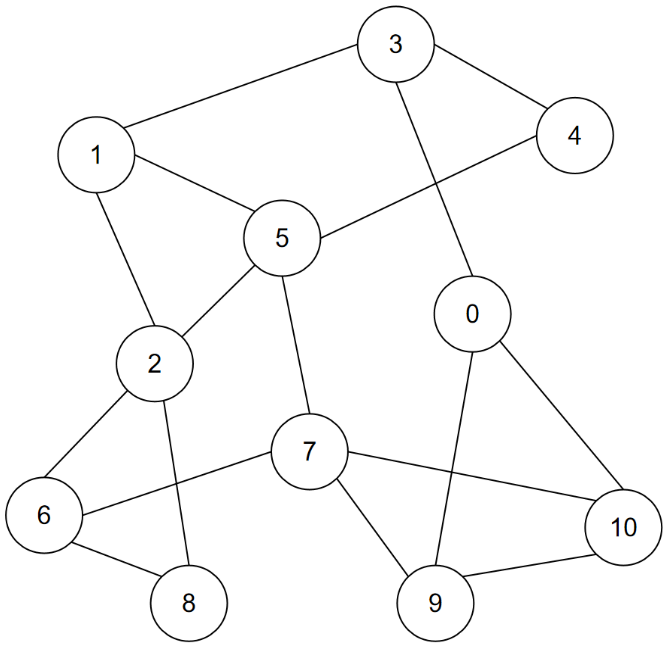
- For written questions, email: yxw2533@case.edu
- For coding problems, email: qat3@case.edu

Written Exercise [50 points]

1.BFS and DFS [20 points]

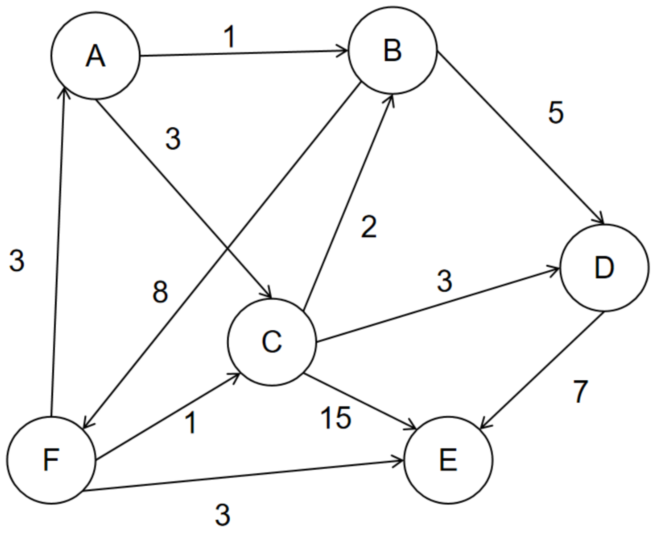
a. (10 pts) Starting at node **0**, list the nodes in order of visitation using breadth-first search (BFS) traversal. When you visit nodes, visit them in ascending numerical order first. Show your work.

b. (10 pts) Starting at node **0**, list the nodes in order of visitation using depth-first search (DFS) traversal. When you visit nodes, visit them in ascending numerical order first. Show your work.



2.Dijkstra’s algorithm [15 points]

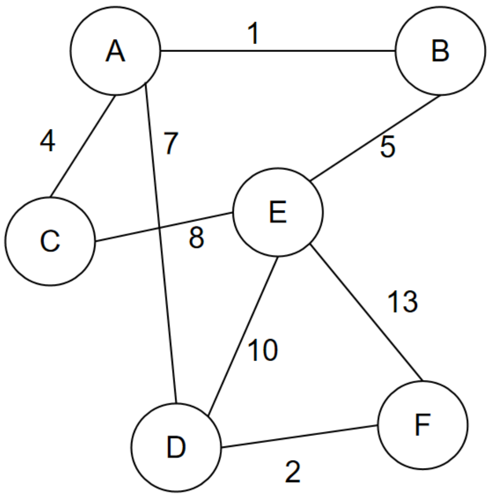
Use Dijkstra's algorithm to find the shortest distance from node A to node E. (10 points). Give the traversal process for each step, including d(node) = distance, the updated distance table. (5 points). Show all work.



3. Adjacency lists and adjacency matrix [10 points]

1.Provide a set of adjacency lists of valid movement through the graph. (5 points)

2.Provide an adjacency matrix for valid movement through the graph. (5 points)



4. Graph Algorithms [5 points]

Draw a simple graph scenario in which Dijksra's algorithm does not return the shortest distance, and explain your diagram.

Programming Exercise [50 points]

In this assignment, you will implement a program to create, manipulate, and query an **undirected, weighted graph** representing a network of cities connected by routes with specified distances. The focus will be on implementing **Dijkstra's algorithm** to find the shortest distance between two nodes and **Prim's algorithm** to compute the minimum spanning tree of the graph.

You will also implement methods for managing the graph, including adding/removing nodes and edges, and displaying the graph in an adjacency list format. Detailed tasks and requirements are outlined below.

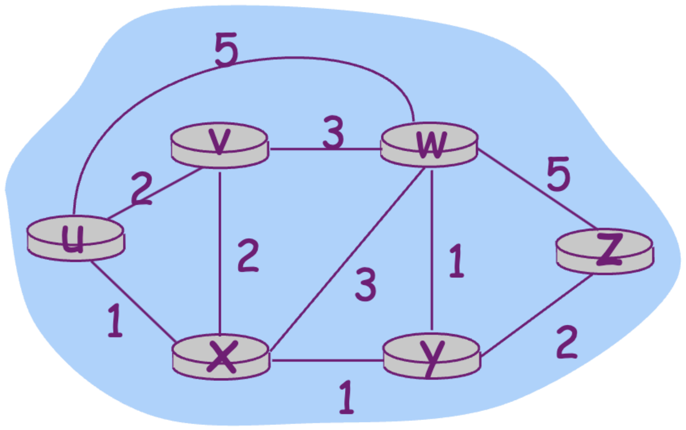
Implementing Graph class

Create a class Graph with the methods listed below. Nodes are referenced by their node names, which are Strings. The methods with boolean return types should return true if successful and false otherwise.

1. **Constructor for Graph class**
   - Graph(): Creates an empty graph and initializes the necessary data structures to store nodes and their connections. The constructor **must** not take any parameters. After creation, the graph object will be able to call all the methods listed in the assignment.

     Example: Graph graph = new Graph();

2. **Constructing undirected, weighted graphs**
    - boolean addNode(String name): Adds a node to the graph. If the node already exists, return false; otherwise, return true.
    - boolean addEdge(String from, String to, int weight): Adds an undirected edge between two nodes with the given weight. If an edge between 2 nodes already exists or if the weight is invalid (e.g., negative), return false.
    - boolean addEdges(String from, String[] toList, int[] weightList): Adds edges from the specified source node from to each node in the toList array, with corresponding weights provided in the weightList array. Returns true if all edges are successfully added. Returns false if the lengths of toList and weightList do not match, if any weight is invalid (e.g., negative).
    - boolean removeNode(String name): Removes the specified node and all its connected edges from the graph. If the node does not exist, return false.
    - boolean removeEdge(String from, String to): Removes the undirected edge between the specified nodes. If either node does not exist or if the edge does not exist, return false.
    - void printGraph(): Prints the graph in the same adjacency list format in the read method described below. The nodes and their neighbors and their neighbors should be listed in alphabetical order. Nodes and their neighbors should be listed in **alphabetical order**. For example, from our class lecture:



Expected output print format:

<nodename1> <weight> <neighbor1> <weight> <neighbor2> ...
<nodename2> <weight> <neighbor1> <weight> <neighbor2> ...

Example printing output:

```
U 2 V 5 W 1 X
V 2 U 3 W 2 X
W 5 U 3 V 3 X 1 Y 5 Z
```

X 1 U 2 V 3 W 1 Y
Y 1 W 1 X 2 Z
Z 5 W 2 Y

- static Graph createGraph(String[][] input): Creates a graph from a 2D array of Strings, where each row represents an edge in the format {"source", "destination", "weight"}. Returns a Graph object. If the input is invalid (e.g., a row does not have exactly 3 elements, the weight is not a valid integer after parsed), return null.

In example, to create the graph above, the method can be called as:
Graph.createGraph(new String[][] { {"U", "V", "2"}, {"U", "X", "1"}, {"V", "W", "3"}, {"V", "X", "2"}, {"U", "W", "5"}, {"W", "Y", "1"}, {"W", "Z", "5"}, {"X", "Y", "1"}, {"Y", "Z", "2"}, {"W", "X", "3"} });

3. **Graph Algorithm**
   - int shortestDistance(String from, String to): Uses Dijkstra's algorithm to compute the shortest distance between two nodes. If either node does not exist or there is no path between them, return -1.

   - List<String[]> minimumSpanningTree(): Uses Prim's algorithm to compute the Minimum Spanning Tree (MST) of the graph. Returns a List<String[]>, where each String[] represents an edge in the format {source, destination, weight}. The edges can be returned in any order. If the graph is disconnected, the method returns null.

     Example output:
     {{"U", "X", "1"}, {"U", "V", "2"}, {"X", "Y", "1"}, {"Y", "W", "1"}, {"Y", "Z", "2"}}

Important Notes
This is the final programming assignment of the course, so try your best to utilize the data structures and algorithms you've learned effectively. You are free to import java.util libraries, create additional classes and helper methods to support your implementation. However, all methods listed in the assignment must be publicly accessible, and their headers **must match exactly** as specified in this document.
Failure to adhere to the requirements, method headers, or return type guidelines will result in **no credit** for the respective implementation. Your submission will be graded based on the following criteria:

- **Compilation**: The program must compile without errors.
- **Effort**: Demonstrates effort and thoughtfulness in the implementation.
- **Correctness**: Produces the correct output for a wide range of inputs.
- **Design**: Uses clean, modular design with appropriate data structures.
- **Efficiency**: Implements algorithms with appropriate time complexity.
- **Encapsulation**: Uses private helper methods and fields where necessary.
- **Testing**: Thoroughly tests each required method using JUnit. A **JUnit test suite** that tests all required methods comprehensively is expected as part of your submission.

Good luck!