Important

There are general homework guidelines you must always follow. If you fail to follow any of the following guidelines you risk receiving a **0** for the entire assignment.

1. All submitted code must compile under **JDK 8**. This includes unused code, so don't submit extra files that don't compile. Any compile errors will result in a 0.
2. Do not include any package declarations in your classes.
3. Do not change any existing class headers, constructors, instance/global variables, or method signatures.
4. Do not add additional public methods.
5. Do not use anything that would trivialize the assignment. (e.g. don't import/use `java.util.ArrayList` for an Array List assignment. Ask if you are unsure.)
6. Always be very conscious of efficiency. Even if your method is to be $O(n)$, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is extremely rare).
7. You must submit your source code, the `.java` files, not the compiled `.class` files.
8. After you submit your files, redownload them and run them to make sure they are what you intended to submit. You are responsible if you submit the wrong files.

Graph Algorithms

For this assignment, you will be coding 4 different graph algorithms. This homework has quite a few files in it, so you should make sure to read ALL of the documentation given to you before asking a question.

Graph Data Structure

You are provided a `Graph` class. The important methods to note from this class are `getVertices`, `getEdges`, and `getAdjList`. `getVertices` provides a Set of `Vertex` objects (another class provided to you) associated with a graph. `getEdges` provides a Set of `Edge` objects (another class provided to you) associated with a graph. `getAdjList` provides a Map that maps `Vertex` objects to Lists of `VertexDistance` objects. This Map is especially important for traversing the graph, as it will efficiently provide you the edges associated with any vertex. For example, consider an adjacency list map where vertex A is associated with a list that includes a `VertexDistance` object with vertex B and distance 2 and another `VertexDistance` object with vertex C and distance 3. This implies that in this graph, there is an edge from vertex A to vertex B of weight 2 and another edge from vertex A to vertex C of weight 3.

Vertex Distance Data Structure

In the `Graph` class and Dijkstra's algorithm, you will be using the `VertexDistance` class implementation that we have provided. In the `Graph` class, this data structure is used by the adjacency list to represent which vertices a vertex is connected to. In Dijkstra's algorithm, you should use this data structure along with a PriorityQueue. At any stage throughout the algorithm, the PriorityQueue of `VertexDistance` objects will tell you which vertex currently has the minimum distance from the source vertex.

Disjoint Set Data Structure (Union-Find)

For Kruskal's algorithm, you will be using the `DisjointSet` class implementation that we have provided to maintain which components are connected. This data structure has two primary functions, `union` and `find`. The idea here is that each disconnected component is maintained as a tree. If two pieces of data are in the same tree, then they are in the same connected component.

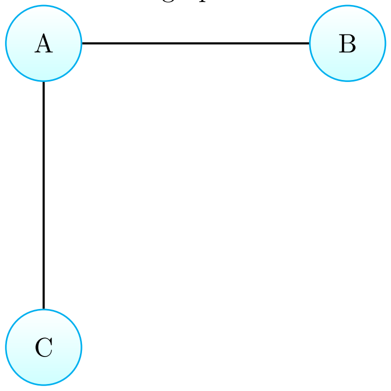
The DisjointSet will begin with all of the data in their own one node trees to begin since nothing is connected originally. As the algorithm progresses, the nodes that are connected in the graph will be linked into the same tree in the DisjointSet.

To see whether or not the two data are in the same tree, it suffices to see if their roots are the same since the root is unique. The `find` method finds the root of the component's tree. Therefore, when checking to see if two vertices are already connected in the MST, you would call `find`

If you add an edge to the MST in Kruskal's, then it will cause two previously disconnected components to become a single component. The `union` method does this for you, merging the two trees representing the two components into a single tree.

Disjoint Set Example

Consider the graph below:



Assume a `DisjointSet` object called `ds` is initialized with the vertices from above. Calling `ds.union(vertexA, vertexB)` joins vertex A and vertex B. Since vertex A and vertex B are in the same component, `ds.find(vertexA).equals(ds.find(vertexB))` returns true. However, calling `ds.find(vertexA).equals (ds.find(vertexC))` returns false since vertex A and vertex C are not in the same component. Calling `ds.union(vertexA, vertexC)` joins vertex C with **both** vertex A and vertex B. Therefore, `ds.find(vertexA) .equals(ds.find(vertexC))` returns true and `ds.find(vertexB).equals(ds.find(vertexC))` returns true.

Search Algorithms

Breadth-First Search is a search algorithm that visits vertices in order of "levels", visiting all vertices one away from the start, then two away, etc. Similar to levelorder traversal in BSTs, it depends on a Queue data structure to work.

Depth-First Search is a search algorithm that visits vertices in a depth based order. Similar to pre/post/in-order traversal in BSTs, it depends on a Stack data structure to work. However, in your implementation, the Stack will be the recursive stack. It searches along one path of vertices from the start vertex and backtracks once it hits a dead end or a visited vertex until it finds another path to continue along. **Your implementation of DFS must be recursive to receive credit.**

Single-Source Shortest Path (Dijkstra's Algorithm)

The next algorithm is Dijkstra's Algorithm. This algorithm finds the shortest path from one vertex to all of the other vertices in the graph. This algorithm only works for non-negative edge weights, so you may assume all edge weights for this algorithm will be non-negative.

There are two main variants of Dijkstra's Algorithm related to the termination condition of the algorithm. The first variant is where you depend purely on the PriorityQueue to determine when to terminate the algorithm. You only terminate once the PriorityQueue is empty. The other variant, the classic variant, is the version where you maintain both a PriorityQueue and a visited set. To terminate, still check if the PriorityQueue is empty, but you can also terminate early once all the vertices are in the visited set. You should implement the classic variant for this assignment. The classic variant, while using more memory, is usually more time efficient since there is an extra condition that could allow it terminate early.

Minimum Spanning Trees (MST - Kruskal's Algorithm)

An MST has two components. By definition, it is a tree, which means that it is a graph that is acyclic and connected. A spanning tree is a tree that connects the entire graph. It must also be minimum, meaning the sum of edge weights of the tree must be the smallest possible while still being a spanning tree.

By the properties of a spanning tree, any valid MST must have $|V| - 1$ edges in it. However, since all undirected edges are specified as two directional edges, a valid MST for your implementation will have $2(|V| - 1)$ edges in it.

Kruskal's algorithm takes in all of the edges of the graph and continually adds the cheapest to the MST as long as that edge does not form a cycle. To handle cycle detection, you will be using a disjoint-set/union-find data structure that we have provided for you.

Self-Loops and Parallel Edges

In this framework, self-loops and parallel edges work as you would expect. These cases are valid test cases, and you should expect them to be tested. However, most implementations of these algorithms handle these cases automatically, so you shouldn't have to worry too much about them when implementing the algorithms.

Grading

Here is the grading breakdown for the assignment. There are various deductions not listed that are incurred when breaking the rules listed in this PDF, and in other various circumstances.

Methods:	
BFS	15pts
DFS	15pts
Dijkstra's	25pts
Kruskal's	20pts
Other:	
Checkstyle	10pts
Efficiency	15pts
Total:	100pts

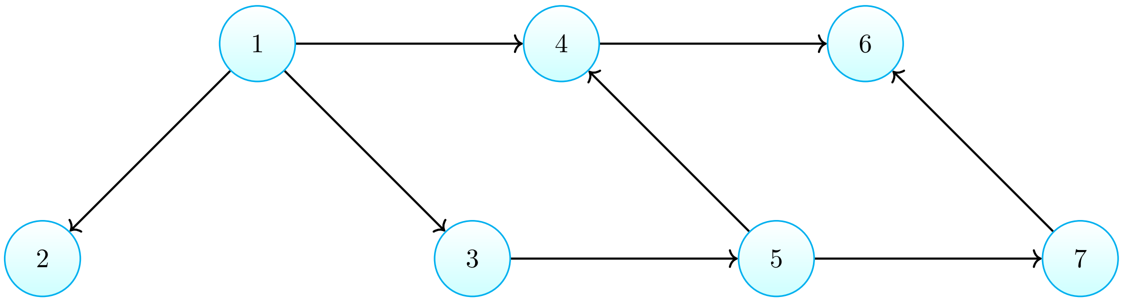
A note on JUnits

We have provided a **very basic** set of tests for your code in `GraphAlgorithmsStudentTests.java`. These tests do not guarantee the correctness of your code (by any measure), nor do they guarantee you any grade. You may additionally post your own set of tests for others to use on the Georgia Tech GitHub as a gist. Do **NOT** post your tests on the public GitHub. There will be a link to the Georgia Tech GitHub as well as a list of JUnits other students have posted on the class Piazza.

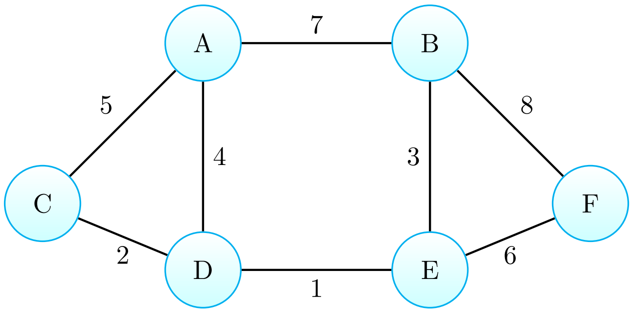
If you need help on running JUnits, there is a guide, available on Canvas under Files, to help you run JUnits on the command line or in IntelliJ.

Visualizations of Graphs

The directed graph used in the student tests is:



The undirected graph used in the student tests is:



Style and Formatting

It is important that your code is not only functional but is also written clearly and with good style. We will be checking your code against a style checker that we are providing. It is located on Canvas, under Files, along with instructions on how to use it. We will take off a point for every style error that occurs. If you feel like what you wrote is in accordance with good style but still sets off the style checker please email Tim Aveni (tja@gatech.edu) with the subject header of "[CS 1332] CheckStyle XML".

Javadocs

Javadoc any helper methods you create in a style similar to the existing Javadocs. If a method is overridden or implemented from a superclass or an interface, you may use `@Override` instead of writing Javadocs. Any Javadocs you write must be useful and describe the contract, parameters, and return value of the method; random or useless javadocs added only to appease Checkstyle will lose points.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies not only to comments/javadocs but also things like variable names.

Exceptions

When throwing exceptions, you must include a message by passing in a String as a parameter. **The message must be useful and tell the user what went wrong**. “Error”, “BAD THING HAPPENED”, and “fail” are not good messages. The name of the exception itself is not a good message.

For example:

Bad: `throw new IndexOutOfBoundsException("Index is out of bounds.");`

Good: `throw new IllegalArgumentException("Cannot insert null data into data structure.");`

Generics

If available, use the generic type of the class; do **not** use the raw type of the class. For example, use `new LinkedNode<Integer>()` instead of `new LinkedNode()`. Using the raw type of the class will result in a penalty.

Forbidden Statements

You may not use these in your code at any time in CS 1332.

- `package`
- `System.arraycopy()`
- `clone()`
- `assert()`
- `Arrays` class
- `Array` class
- `Thread` class
- `Collections` class
- `Collection.toArray()`
- Reflection APIs
- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

If you’re not sure on whether you can use something, and it’s not mentioned here or anywhere else in the homework files, just ask.

Debug print statements are fine, but nothing should be printed when we run your code. We expect clean runs - printing to the console when we’re grading will result in a penalty. If you submit these, we will take off points.

Provided

The following file(s) have been provided to you. There are several, but you will edit only one of them.

1. `GraphAlgorithms.java`

   This is the class in which you will implement the different graph algorithms. Feel free to add private static helper methods but **do not add any new public methods, new classes, instance variables, or static variables**.

2. `GraphAlgorithmsStudentTests.java`

   This is the test class that contains a set of tests covering the basic operations on the `GraphAlgorithms` class. It is not intended to be exhaustive and does not guarantee any type of grade. Write your own tests to ensure you cover all edge cases. The graphs used for these tests are shown above in the pdf.

3. `Graph.java`

   This class represents a graph. **Do not modify this file.**

4. `Vertex.java`

   This class represents a vertex in the graph. **Do not modify this file**.

5. `Edge.java`

   This class represents an edge in the graph. It contains the vertices connected to this edge and its weight. **Do not modify this file**.

6. `DisjointSet.java`

   This class represents a union-find data structure to be used for Kruskal's algorithm, consisting of the operations find and union. **Do not modify this file**.

7. `DisjointSetNode.java`

   This class represents a node for `DisjointSet.java`. **Do not modify this file**.

8. `VertexDistance.java`

   This class holds a vertex and a distance together as a pair. It is meant to be used with Dijkstra's algorithm. **Do not modify this file.**

Deliverables

You must submit **all** of the following file(s). Please make sure the filename matches the filename(s) below, and that *only* the following file(s) are present. If you make resubmit, make sure only one copy of the file is present in the submission.

After submitting, double check to make sure it has been submitted on Canvas and then download your uploaded files to a new folder, copy over the support files, recompile, and run. It is your responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `GraphAlgorithms.java`