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. For example, do not add `throws` to the method headers since they are not necessary.
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 `ArrayList` 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. Only the last submission will be graded. Make sure your last submission has **all** required files. Resubmitting will void all previous submissions.
9. 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 code 4 different graph algorithms. This homework has many files included, so be sure to read ALL of the documentation given before asking questions.

Graph Data Structure

You are provided a `Graph` class. The important methods to note from this class are:

- `getVertices` returns a Set of `Vertex` objects (another class provided to you) associated with a graph.
- `getEdges` returns a Set of `Edge` objects (another class provided to you) associated with a graph.
- `getAdjList` returns 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 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. When utilizing `VertexDistance` in this algorithm, the vertex attribute should represent the destination vertex and the distance attribute should represent the minimum cumulative path cost from the source vertex to the destination vertex.

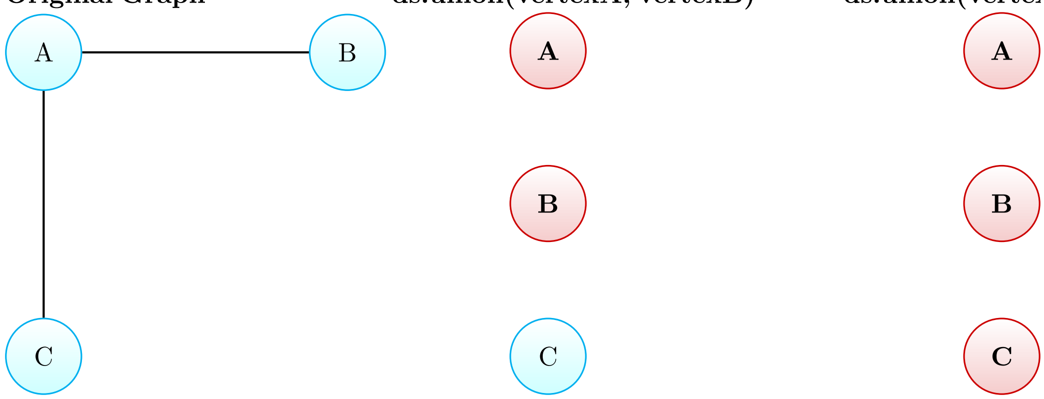
Disjoint-Set Data Structure

In Kruskal's algorithm, you will be using the `DisjointSet` class implementation that we have provided. You should use this data structure to determine whether vertices are already connected by a path (which means adding an edge between them would create a cycle) and to merge sets of edges together. These methods are find(...) and union(...) respectively.

Disjoint Set Example

Consider the graph below:

Original Graph **ds.union(vertexA, vertexB)** **ds.union(vertexA, vertexC)**



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 (BFS, DFS)

Breadth-First Search is a search algorithm that visits vertices in order of "level", visiting all vertices one edge away from start, then two edges away from start, 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. 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. In order to keep track of the cumulative distance from the source vertex to the vertices you visit in this algorithm, you will need to use the `VertexDistance` data structure we are providing you. At any stage throughout the algorithm, the PriorityQueue of `VertexDistance` objects will tell you which vertex currently has the minimum cumulative distance from the source vertex.

There are two commonly implemented terminating condition variants for Dijkstra's Algorithm. The first variant is where you depend purely on the PriorityQueue to determine when to terminate. 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 to terminate early.

Minimum Spanning Trees (Kruskal's Algorithm)

A tree is a graph that is acyclic and connected. A spanning tree is a subgraph that contains all the vertices of the original graph and is a tree. An MST has two main qualities: being minimum and a spanning tree. Being minimum dictates that the spanning tree's sum of edge weights must be minimized.

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.

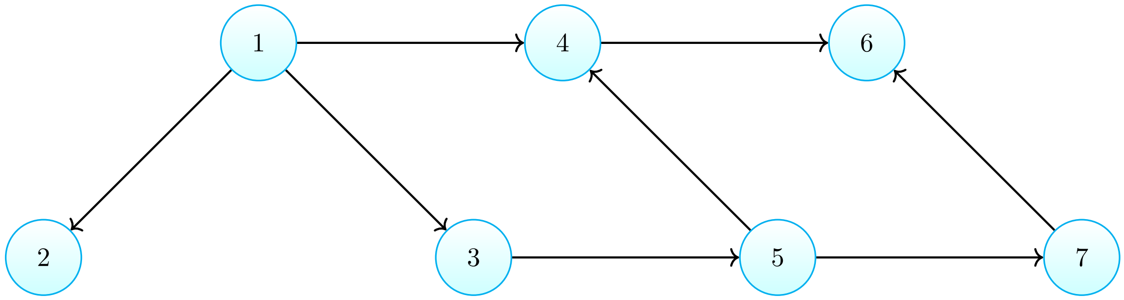
Kruskal's algorithm builds the MST using a Disjoint-Set data structure. This is a greedy algorithm, and at each step, the algorithm adds the cheapest edge in the entire graph that does not cause a cycle. Cycle detection is done with a Disjoint-Set. If an edge connects vertices that are in the same set, then the algorithm continues to the next candidate edge. Unlike the previous algorithm, Dijkstras, Kruskal's algorithm does not require the use of the `VertexDistance` data structure since it does not begin at a source vertex. Instead, it greedily selects edges with the lowest path costs until an MST is formed for each connected component.

Self-Loops and Parallel Edges

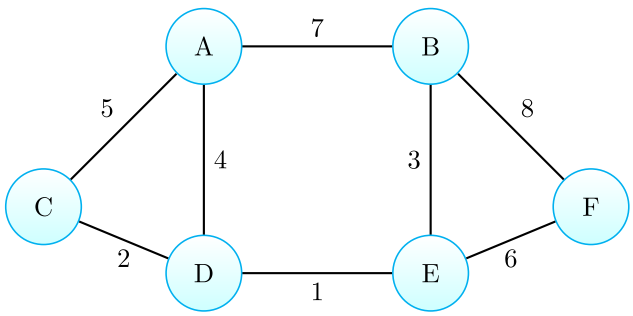
In this framework, self-loops and parallel edges work as you would expect. If you recall, self-loops are edges from a vertex to itself. Parallel edges are multiple edges with the same orientation between two vertices. In other words, parallel edges are edges that are incident on precisely the same vertices. 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.

Visualizations of Graphs

The directed graph used in the student tests is:



The undirected graph used in the student tests is:



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

JUnits

We have provided a **very basic** set of tests for your code. 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.

Collaboration Policy

Every student is expected to read, understand and abide by the Georgia Tech Academic Honor Code.

When working on homework assignments, you **may not** directly copy code from any source (other than your own past submissions). You are welcome to collaborate with peers and consult external resources, but you **must** personally write all of the code you submit. **You must list, at the top of each file in your submission, every student with whom you collaborated and every resource you consulted while completing the assignment**.

You may not directly share any files containing assignment code with other students or post your code publicly online. If you wish to store your code online in a personal private repository, you can use Github Enterprise to do this for free.

The only code you may share is JUnit test code on a pinned post on the official course Piazza. Use JUnits from other students at your own risk; **we do not endorse them**. See each assignment's PDF for more details. If you share JUnits, they **must** be shared on the site specified in the Piazza post, and not anywhere else (including a personal GitHub account).

Violators of the collaboration policy for this course will be turned into the Office of Student Integrity.

Style and Formatting

It is important that your code is not only functional, but written clearly and with good programming style. Your code will be checked against a style checker. The style checker is provided to you, and is located on Canvas. It can be found under Files, along with instructions on how to use it. A point is deducted for every style error that occurs. If there is a discrepancy between what you wrote in accordance with good style and the style checker, then address your concerns with the Head TA.

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 LinkedList<Integer>()` instead of `new LinkedList()`. 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 we've noted the ones to edit.

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. `Graph.java`

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

3. `Vertex.java`

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

4. `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**.

5. `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.**

6. `DisjointSet.java`

   This class represents a Disjoint-Set. It has the operations `union` and `find`. It is meant to be used with Kruskal's algorithm. **Do not modify this file.**

7. `DisjointSetNode.java`

   This class represents an element in a Disjoint-Set. It is meant to be used with Kruskal's algorithm. **Do not modify this file.**

8. `GraphAlgorithmsStudentTests.java`

   This is the test class that contains a set of tests covering the basic algorithms in 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.

Deliverables

You must submit **all** of the following file(s). Make sure all file(s) listed below are in each submission, as only the last submission will be graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present. The only exception is that Canvas will automatically append a `-n` depending on the submission number to the file name(s). This is expected and will be handled by the TAs when grading as long as the file name(s) before this add-on matches what is shown below. If you resubmit, be sure only one copy of each file is present in the submission. If there are multiple files, do not zip up the files before submitting; submit them all as separate files.

Once submitted, double check that it has uploaded properly on Canvas. To do this, download your uploaded file(s) to a new folder, copy over the support file(s), recompile, and run. It is your sole responsibility to re-test your submission and discover editing oddities, upload issues, etc.

1. `GraphAlgorithms.java`