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 11**. 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. You may create helper methods, but any helper method you create should be **private**.
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 consider the efficiency of your code. Even if your method is *O(n)*, traversing the structure multiple times is considered inefficient unless that is absolutely required (and that case is very rare).
7. You must submit your source code - the .java files. Do not submit compiled code - the .class files.
8. Only the last submission will be graded. Make sure your last submission has all required files. Resubmitting voids prior submissions.

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 it located on Canvas. A point is deducted for every style error that occurs. Please double check before you submit that your code is in the appropriate style so that you don't lose any unnecessary points!

Javadocs

Javadocs should be written for any private helper methods that you create. They should follow a style similar to the existing javadocs on the assignment. 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.

Additionally, make sure you include your name, version, user ID, and GT ID in any file submitted to Gradescope.

Vulgar/Obscene Language

Any submission that contains profanity, vulgar, or obscene language will receive an automatic zero on the assignment. This policy applies to all aspects of your code, such as comments, variable names, and javadocs.

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", "Oof - Bad things are happening", and "FAIL" are *not* good messages. Additionally, the name of an exception itself is not a good message.

In addition, you may not use try catch blocks to catch an excpetion unless you are catching an exception you have explicitly thrown yourself with the **throw new ExceptionName('Exception Message");** syntax

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 any of the following in your code at any time in CS 1332. If you are not sure whether you can use something, and it is not explicitly listed here, just ask. Debug print statements are fine, but should be either removed or commented out prior to submission. If print statements are left in, assignments are messy to grade, and checkstyle points will be deducted.

- package
- System.arraycopy()
- clone()
- assert()
- Arrays class
- Thread class
- Collections class
- Collection.toArray()
- Refleciton APIs

- Inner or nested classes
- Lambda Expressions
- Method References (using the :: operator to obtain a reference to a method)

JUnits

We have provided a **very basic** set of tests for your code. These tests do not guarantee the correctness of your code, 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.

Deliverables

You must submit all of the following file(s) to the corresponding assignment on Gradescope. Make sure all file(s) listed below are in each submission, as only the last submission is graded. Make sure the filename(s) matches the filename(s) below, and that *only* the following file(s) are present.

1. GraphAlgorithms.java

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.

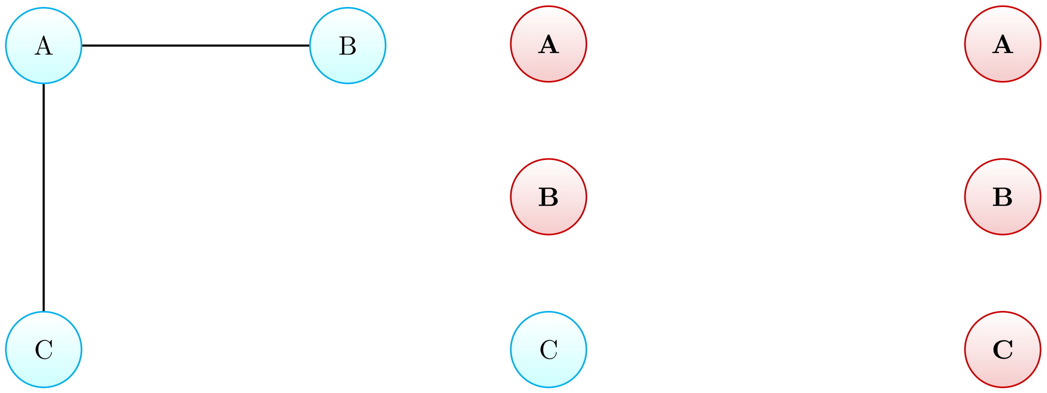
Disjoint-Set Data Structure

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 must 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.

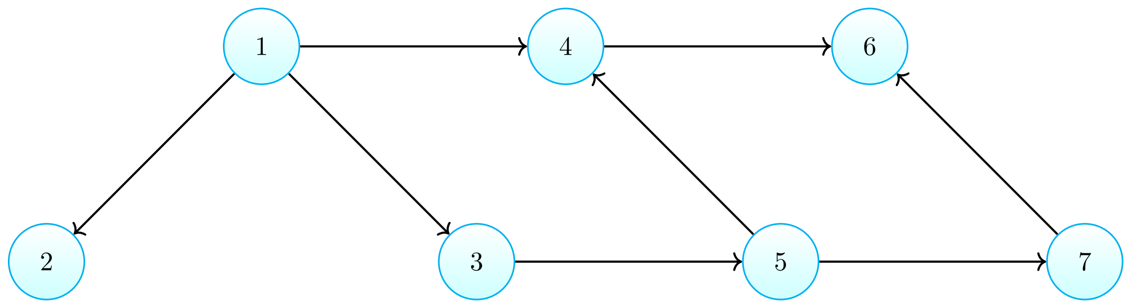
Kruskal's algorithm builds the MST using a Disjoint-Set data structure. This is a greedy algorithm, and at each step, the algorithm adds the cheapest edge in the entire graph that does not cause a cycle. Cycle detection is done with a Disjoint-Set. If an edge connects vertices that are in the same set, 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

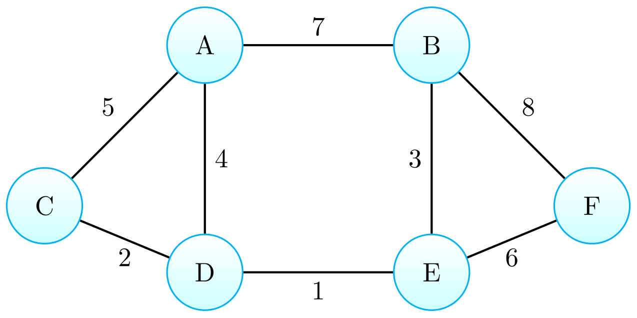
In this framework, self-loops and parallel edges work as you would expect. If you recall, self-loops are edges from a vertex to itself. Parallel edges are multiple edges with the same orientation between two vertices. In other words, parallel edges are edges that are incident on precisely the same vertices.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

- GraphAlgorithms – 90 pts
- Checkstyle – 10 pts
- Total: 100 pts

Provided

The following file(s) have been provided to you. There are several, but we've noted the ones to edit.

These are the classes to be implemented. Feel free to add private helper methods but **do not add any new public methods, inner/nested classes, instance variables, or static variables**.

1. `GraphAlgorithms.java`

These are files that represent the structures needed to implement the algorithms. **Do not alter these files.**

1. `Graph.java`
2. `Vertex.java`
3. `VertexDistance.java`
4. `Edge.java`
5. `DisjointSet.java`
6. `DisjointSetNode.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.**

1. `GraphAlgorithmsStudentTest.java`