1 Prologue

1. To install, type `cs0160_install <project_name>` into a shell. The script will create the appropriate directories and deposit stencil files into them.

2. To compile your code, type `make` in your project directory. To run your code and launch the visualizer, type `make run` in the same directory.

3. To run tests, run `make run_tests` from your project directory.

4. To hand in your project, go to the directory you wish to hand in, and type `cs0160_handin <project_name>` into a shell.

5. To run a demo of this project, type `cs0160_runDemo <project_name>` into a shell.

6. The NDS4 (you'll be using some data structures from this package) documentation can be found here: [http://cs.brown.edu/courses/cs016/static/files/docs/nds4/index.html](http://cs.brown.edu/courses/cs016/static/files/docs/nds4/index.html). This is also linked off of the class website.

7. Remember to not include any identifying information in your hand in.

2 Introduction

2.1 Silly Premise

Carl is reaching that ripe old age where he's getting a little nutty. Part of his charm is an affinity for floating away in his house and dreaming up new communities in the most unlikely of places. Carl is off his rocker, folks. Bored one day in his floating house, while snacking on dog food, Carl looks down at the forest below and decides that he wants to build a treehouse network that makes getting from tree to tree as efficient as possible. Lucky for Carl, he's a tech whiz, too. He has floated near a cell tower and called you to implement two algorithms that will help him create his minimum spanning kingdom (MSK).

2.2 Serious Premise

In this assignment, you will implement a graph, as well as the Kruskal and Prim-Jarnik algorithms for finding minimum spanning forests (MSFs) of a graph.
3 Overview of Your Tasks

We have provided stencil code for the following four classes: `AdjacencyMatrixGraph`, `MyKruskal`, `MyPrimJarnik`, and `MyDecorator`. You need to fill in these stencils. We will provide a brief overview and specific information about these classes and their methods in Section 6 of this handout.

1. **AdjacencyMatrixGraph**: You will implement your graph here. Its underlying data structure will be an adjacency-matrix.

2. **MyKruskal**: In this class, you will implement Kruskal’s algorithm for `Graph` classes. You will use *decorations* (see Section 6.6) in this class to mark vertices with specific information. Your algorithm should return a collection (any class that implements `java.util.Collection`) of the edges in your minimum spanning forest (MSF).

3. **MyPrimJarnik**: Here, you will implement the Prim-Jarnik algorithm. Again, you will use decorations to mark vertices with specific information, and should return a collection of the edges in your MSF.

4. **MyDecorator**: Here, you will define methods that allow you to “decorate” vertices of your graph with specific information.

3.1 Extra Credit: Edge Set Graph

For extra credit, you can implement an Edge Set Graph, in addition to the required Adjacency Matrix Graph. The specifics regarding how you should approach this is defined in [Section 10](#).

4 Reading

Refer to Chapter 5.1 of [Dasgupta](#) for readings on Prim-Jarnik, Kruskal, and Union-Find.

5 Visualizer

For the most part, using the visualizer is intuitive: to create a vertex, left-click. To create two vertices with an edge between them, click and drag landing either on another, pre-existing vertex or on nothing to create a new one. Here are some less intuitive controls:

- **select a vertex v as v1**: left-click on the vertex
- **select a vertex v as v2**: shift + click on the vertex (or option-click for some computers)
- **remove a vertex and any incident edges**: right-click on the vertex
• **select an edge:** left-click on the edge

• **remove an edge:** right-click on the edge

A note on adding edges: You can choose to add edges with a random weight, with the
distance between two vertices as the weight, or with custom weights that you give as input
to the visualizer.

The following is a list of methods of your graph that correspond to functionality in the
visualizer:

• `areAdjacent(v1, v2)`: returns a boolean value describing whether the two selected
vertices, `v1` and `v2`, are connected by an edge.

• `connectingEdge(v1, v2)`: if the two selected vertices `v1` and `v2` are connected by
an edge, returns the name of the edge.

• `endVertices(e)`: returns the two endpoints of an edge `e`.

• `opposite(v1, e)`: returns the second endpoint of an edge `e` (that is: if you select a
vertex and a connected edge - this will return the *other* vertex attached to that edge).

• `incidentEdges(v1)`: returns all edges attached to the selected vertex `v1`.

Note that in order for the visualizer to work at all, you need to implement the fol-
lowing methods: `vertices()`, `insertVertex()`, `vertices()`, `edges()`, `insertVertex()`,
`insertEdge()`, `endVertices()`.

### 5.1 Other Visualizer Functionality

1. The visualizer allows you to save and load graph files. This will help for testing. More
on this in Section 9.

2. The visualizer has optional functionality that will allow you to animate your MSF-
finding algorithms. Note: *This is not required*. However, we highly encourage it—
the code is fairly simple, and it will make debugging a lot easier for you.

   There are two types of animations available to you: edge animation and cloud anima-
tion. If you correctly implement the visualization animation, a slider will appear on
the visualizer after you have calculated a MSF. You can move the slider to show you
the creation of your MSF over time—the edge animation will show you the order in
which the edges were added, while the cloud animation will show you which vertices
belong to which clouds as Kruskal’s algorithm progresses. For more on implementing
animation, see Section 6.6.2.

   **Note:** Edge animation will work for either Prim-Jarnik or Kruskal, while cloud an-
imation will only work for Kruskal. You can implement the edge animation without
implementing the cloud animation, but not vice versa.
6 Your Code

6.1 Stencil

The stencil contains descriptions of the methods you’ll write, their run-time requirements, parameters, and return values. Unless otherwise noted, all functions in the support code are O(1).

6.2 AdjacencyMatrixGraph Class

6.2.1 Description

The underlying data structure for your graph will be an adjacency matrix. An adjacency matrix is a 2D array. The \(i\)th column and row of this array represents the \(i\)th vertex in the graph. So if vertex 1 and 5 are connected by an edge, then entries (1, 5) and (5, 1) in the adjacency matrix contain the edge that connects vertices \(i\) and \(j\). Otherwise they are null. Note that every edge is in two places in the array.

Suppose we have vertices \(v_1\), \(v_2\), and \(v_3\), and that edge \(e_1\) connects \(v_1\) and \(v_2\), and that edge \(e_2\) connects vertices \(v_1\) and \(v_3\). Here’s the corresponding adjacency matrix:

\[
\begin{array}{ccc}
  v_1 & v_2 & v_3 \\
  v_1 & \text{null} & e_1 & e_2 \\
  v_2 & e_1 & \text{null} & \text{null} \\
  v_3 & e_2 & \text{null} & \text{null} \\
\end{array}
\]

6.2.2 Implementing the Adjacency Matrix

Since we are using an adjacency matrix as our graph implementation, each vertex must have a “number”, so that it can represent an index of a row and column in the array. This assignment is not as trivial as it may appear. Arrays have a fixed size, so you cannot indefinitely increase the number for each new vertex because you will exceed the size of your array. Note: the number associated to a given vertex must be unique - it must not be associated to any other vertex.

Your array should be able to hold up to \(\text{MAX\_VERTICES}\) vertices, which is a constant defined in the support code.

6.3 Vertex and Edge Attributes

You will need to keep track of some information about the vertices and edges of your graph. To that end, the support code classes \texttt{GraphVertex} and \texttt{GraphEdge} have some methods that you may find helpful. View the Javadocs.
Note: In the support code, you’ll notice there are references to `CS16Vertex` and `GraphVertex`. `CS16Vertex` is an interface that the class `GraphVertex` implements. You should be declaring vertices of type `CS16Vertex` but instantiating new `GraphVertexes`.

### 6.4 MyKruskal Class

#### 6.4.1 General Description

This class implements a slightly modified version of Kruskal’s MST algorithm. In this implementation, Kruskal’s algorithm has been extended to calculate the MST of each connected subgraph of your graph. That is, given a disconnected graph, your algorithm will compute a the Minimum Spanning Forest (MSF) rather than a single MST. The algorithm is required to run in $O(|E| \log |V|)$ time. Do the reading in Section 4 to get a full description of the algorithm.

Your implementation of Kruskal’s algorithm will have three phases:

1. Each vertex of the graph is put into its own “cloud.” Initially, there will be $|V|$ clouds.

2. Each edge of the graph is inserted into a priority queue, with its weight as the key.
   You should use our implementation of a heap adaptable priority-queue:
   `support.graph.CS16AdaptableHeapPriorityQueue`.

3. The edge with minimum weight is removed from the queue. If its end vertices belong to different clouds, the clouds are merged. This step is repeated until there are no unexamined edges remaining. In order for cloud-merging to run in just over amortized $O(1)$ time, you will need to use Union-Find, which is discussed in greater detail in Section 6.4.2.

4. Return a collection of the edges that are in the final MSF. You may add edges to the collection as you go along, or you may complete the entire algorithm and then check to see which edges you should add to the collection. Feel free to return the edges in any sensible data structure which implements the `java.util.Collection` interface.

#### 6.4.2 Union-Find

In order to meet the runtime requirement for Kruskal’s algorithm, you will be implementing a Union-Find data structure. You can read about Union-Find in the *Algorithms* textbook mentioned above in Section 4 as well as on Wikipedia. Here is a general overview of the algorithm (with some tips for your specific implementation of it):

- Vertices belong to different “clouds”. In one implementation of the cloud structure, you can use decorations to label each vertex with its own cloud, which makes it $O(1)$ to look up which cloud a given vertex belongs to. However, this makes merging clouds expensive: if you merge two clouds, you will have to re-label all of the vertices in one...
cloud with the other cloud’s number. So instead of labeling every vertex with its cloud, think of clouds as being mini “trees” in your graph. You would then proceed as follows:

– The root of the tree will know to which cloud it belongs, while other vertices will need to ask the root what cloud it belongs to in order to determine their own clouds.

– At the beginning of Kruskal’s algorithm, all vertices are in their own clouds. After the first step, you will combine two vertices into one cloud. For example, suppose A is in cloud 1, and B is in cloud 2. Instead of re-labeling vertex B as cloud 1, simply make vertex B “point” to vertex A.

– When you want to find the cloud of vertex B, you can no longer just ask it what its cloud is. However, you can ask it what its parent’s cloud is: you will follow the parent pointer to vertex A, which is labeled as cloud 1.

• This simple way of doing Union-Find runs in amortized $O(|E| \log |E|)$ time. The next points explain something called “path compression,” which makes Union-Find run in just over amortized $O(1)$ time. Because you’re a super-awesome Brown C.S. student, you’ll be implementing this more efficient method.

– If we “join“ the vertices into one tree, we end up with something that looks like $D \rightarrow C \rightarrow B \rightarrow A$? In the simple method, whenever you want to look up the cloud to which D belongs, you have to chase all the way up the tree from D to A.

– Here’s the trick: only do the work of searching for D’s cloud once. As you follow the chain of parent pointers to A—the root, and only vertex which knows what cloud it’s in—you can reset the parent pointers of D and C to directly point to A. Then, next time you want to look up D’s cloud (or C’s cloud, for that matter), you’ll have O(1) lookup.

– But you’re not quite done: when two clouds are merged, you still need to determine which cloud should get merged into the other to optimize run time. To do this, each root should have a property called “rank”: for clouds of size one, the rank of the root of the cloud is 0. For all clouds of size greater than one, rank is updated during a `union()` operation: when merging two clouds of the same rank, arbitrarily make one root point to the other and increment the rank of the root of the new, merged, cloud by 1. When merging two clouds of different rank, make the lower-ranked cloud’s root point to the higher-ranked cloud’s root, and give the merged cloud the rank of the higher-ranked cloud.

When actually implementing Union-Find for this project, keep these issues in mind (and think about how the MyDecorator class could help you):

• How will you keep track of cloud numbers and parent references?
• How will you keep track of the rank of a cloud in a way that meets the runtime requirements?

6.5 **MyPrimJarnik Class**

This class implements a slightly modified version of the Prim-Jarnik MST algorithm. Like the Kruskal implementation, the Prim-Jarnik algorithm has been extended to calculate a MSF (the MST of each connected subgraph of your graph). This algorithm is required to run in $O(|V|^2 \log |V|)$ time. Use the same heap you used in Kruskal, `CS16AdaptableHeapPriorityQueue` (don’t use the NDS4 one).

Here’s an outline of the algorithm (which is presented in more depth in the reading – Section 4).

1. First, decorate each vertex with a key-value: you can think of these values as the “cost” of adding a given vertex to the MSF. At each iteration of the algorithm, you will want to add the edge that connects to the cheapest vertex to the MSF. Initially, each node starts with a value of “infinity.” (You can use the `Integer.MAX_VALUE` constant to represent infinity.)

2. Insert the vertices into a priority queue, using the keys that were assigned in the previous step.

3. Remove the minimum vertex ($v$) from the priority queue, and add the edge that most recently updated it (if any) to the MSF.

4. Examine all (if any) of $v$’s incident edges $e$ whose opposite vertex $u$ remains unvisited. If $u$ has a key greater than the weight of $e$, update the key to become the weight of $e$ (and make sure this change is reflected in the priority queue—to do this, you’ll need to know the `Entry` in the PQ associated with each vertex). Keep track of which edge most recently updated the key of vertex $u$ (you may find a decoration useful here).

5. Continue to remove vertices from the priority queue (adding edges to the MSF and updating other vertices as you go along, as described in steps 3 and 4) until the PQ is empty.

Note that in addition to decorating edges as you go along, you must ultimately return a collection of the edges that are in the final MSF. You may add edges to the collection as you go along, or you may complete the entire algorithm and then check to see which edges you should add to the collection. Feel free to use any sensible implementation of the `java.util.Collection` interface.

---

1It can be made more efficient—it would run with complexity $O(|V| + |E| \log |V|)$ if we were to use a binary heap and an adjacency list and with complexity $O(|E| + |V| \log |V|)$ if we were to use a Fibonacci heap and an adjacency list. Our implementation uses a binary heap and an adjacency matrix, which gives us runtime $O(|V|^2 \log |V|)$.
6.6  MyDecorator Class

6.6.1  Description

The `CS16Decorator` interface represents “decorations” that you will use to label vertices and edges with specific information. The `MyDecorator` class is the implementation of this interface that you will write and use. You may want to use this class to note that a specific edge is part of your minimum spanning forest, or that a specific vertex is in a specific cloud during Kruskal’s algorithm. Think about what type of data structure you want to use in this class; it should be a data structure that you can quickly (i.e., constant-time association and lookup) use to associate edges and vertices with some type of information.

If you want to use the visualizer animation functionality, you will need to use the decorations discussed in the Section 6.6.2.

6.6.2  Implementing Visualizer Animation

If you choose to animate your algorithms, our support code requires that your code include some specific types of decorations.

1. How to implement edge animation:
   - Include a decoration that maps edges of your graph to Boolean values, which represent whether a given edge is currently in the MSF.
   - Inform the visualizer every time it needs to update MSF edges. To do this, call `addEdgeAnimation` on the instance of the `CS16GraphVisualizer` passed in as a parameter to your `getMinSpanForest(...)` methods and pass your decoration as an argument. You must call this method every time you update one of the `edge → Boolean` decorations.

2. How to implement cloud animation:
   - Include a decoration that maps vertices of your graph to Integers; this decoration should represent the cloud number to which a given vertex currently belongs. (Note that only vertices which are at the “root” of a cloud should have a cloud decoration; see Section 6.4.2 for more information.)
   - Include a decoration that maps a vertex of your graph to another vertex; this decoration should represent the “parent” of a given vertex in the Union-Find structure. (In this case, only “non-root” vertices should have a parent decoration; see Section 6.4.2 for more information.)
   - Inform the visualizer every time it needs to update its set of clouds. To update the clouds, call `addCloudAnimation` on the instance of the `CS16GraphVisualizer` that is passed in as a parameter to your `getMinSpanForest(...)` methods, and pass your cloud decoration and parent decoration as arguments.
For your animation to work correctly, you must call this method every time you update one of the vertex → cloud or vertex → vertex decorations.

7 Using Eclipse

If you would like to use eclipse, you may certainly do so. In order to set up your project and make eclipse work with the support code, you’ll need to do the following:

- Select File → New → Java Project
  - Enter “<project_name>”, lowercased, for example ”heap”, for the project name.
  - Un-check the box saying “Use default location” and in the “Location” box, enter /gpfs/main/home/<your-login>/course/cs0160/<project_name>
  - Click “next”
  - Under the “libraries” tab choose “Add External JARs...”
    - Select /course/cs0160/lib/cs0160.jar
    - Select /course/cs0160/lib/nds4/nds4.jar
    - Select /course/cs0160/lib/junit-4.12.jar
    - Select /course/cs0160/lib/hamcrest-core-1.3.jar
  - Click “Finish”
    - If it isn’t already made for you, use File → New → Source Folder to create a new source folder in your new project named “src”
    - Use File → New → Package to create a new package in your new source folder named, for example heap, and move all the stencil java files into this package. Ignore any errors.

- Right-click on App.java and select Run As → Java Application. Now you can run your program by pressing the green play button at the top of your screen and selecting “Java application” if prompted

- To run the tests in eclipse, you can right-click on TestRunner.java and click Run As → Java Application.

- Alternatively, if you want to run one test file, you can right-click on that file and select Run As → JUnit test

- To configure your Eclipse projects to run over FastX or SSH, follow these setup steps
  - Right click on the package icon next to the project name. Go to properties.
  - Go to Run/Debug Settings, select the main window App and click Edit.
  - Go to the arguments tab and, and enter -Dprism.order=sw in the VM arguments block
Hit Apply and OK
You should be all set to work on this project remotely with Eclipse. Make sure to do this for each new project.

8 Working from Home

If you wish to work locally, you should be able to set up the project in Eclipse on your home computer by following the directions that are listed in the ‘Using Eclipse’ section of the handout. You will also need to copy the image at “/course/cs016/lib/graph-images/Dragon.jpg”, put it in the same directory as your code on your local computer, and update the image path in the App.java file.

Note that you will need to have the support libraries listed in that section copied onto your local computer in order to reference them – you can copy the libraries from department machines using SFTP or SCP commands, provided that you have SSH set up.

If you are unfamiliar with using SFTP or SCP commands, please see a TA or Sunlab Consultant for assistance.

As always, be sure to test your code on department machines before turning it in!

9 Testing

As per usual, we require a well thought-out set of tests. We won’t give you too many hints - you’ve come a long way in CS16, so you’re a pro at this by now! You will write these test functions in the stencil files that you are provided: GraphTest.java, and MsfTest.java. As always, please comment each testing function extensively. If there’s anything particularly notable about your tests, include an explanation in your README.

10 Extra Credit Edge Set Graph

Your EdgeSetGraph, to be filled in inside EdgeSetGraph.java, should have the same functionality as your AdjacencyMatrixGraph, as a class that implements the same Graph interface. The runtime requirements for each of the methods is written in the method comments. You can receive at most 10 points of extra credit.

10.1 Basic walkthrough of functionality

The lecture slides explain how various functions of an edge set achieve the runtime specifications they do, so please read through those slides to gain an overall understanding of how an edge set works first.
10.2 EdgeMap (Big Hint!)

In this implementation of an EdgeSet, we’ll actually be maintaining a map (Like a dictionary or hashtable, a way of mapping keys to values) in a variable called _edges. When you’re adding a particular key (edge) to this map, the value should be the same edge as the key. Wait what? Why wouldn’t we use a set instead, where the keys and the values are the same?

We take this approach to make the connectingEdge(v1, v2) method run in O(1), as described below.

Suppose that we had, at some point in the past, added an Edge with v1 = A, v2 = B, element = 100 to our _edges. Given just the variables v1 and v2, how could we extract the real edge that connects the two? We can’t look through all of the edges to find one where the adjacent vertices are v1 and v2, as that would run in O(|E|) time.

Instead, we can commit something similar to identity theft. Instances of the GraphEdge class are considered equal as long as they have the same values for their FromVertex and ToVertex. That is, these two edges are considered ‘equal’

E1 = v1 = A, v2 = B, element = 100
E2 = v1 = A, v2 = B, element = 50

But these two are not.

E1 = v1 = B, v2 = A, element = 100
E2 = v1 = A, v2 = B, element = 50

Thus, in order to ‘get’ (hint hint) the real edge that we had stored in _edges, we can ask for that value using a new, ‘fake’ Edge with the corresponding v1 and v2 values as the key!

10.3 Graph Un-directedness

Consider that for this assignment, all graphs that we make are undirected. So connectingEdge(A, B) and connectingEdge(B, A) must always return the same value. There are a number of ways to handle this, so give this some thought!

11 What to Hand In

1. Code for the four classes AdjacencyMatrixGraph, MyKruskal, MyPrimJarnik, MyDecorator, and optionally EdgeSetGraph.
2. Code for the two testing classes, GraphTest and MsfTest.
3. A README pointing out any bugs in your code, any significant design choices (talk about the types of decorations you used, and how/why you used them), and descriptions of your test functions.