COMP 282 - Project 2

Fall 2018

Instructions for Submission

You must provide all source files in a single `.zip` file. This file must contain no directories whatsoever. That is, if I were to unzip it, the contents would be only the `.java` files required for the project. For example, in a *nix environment, I would create a submission by navigating to my source directory issuing a `zip AdamClark *.java`. **Under no circumstances will a submission be accepted if it is the product of you zipping your IDE's project directory!**

There are to be absolutely no packages used in your submissions. The use of the default package is, generally, not advised for enterprise-level work; this, however, is not that.

Additionally, you should not include any `main` functions in your submission, only those files required for each project part are needed.

The following files **must** be present in your submission:

- `Graph.java`

A series of "checker" functions have been provided for each part of the project. Ensure your project passes these basic checks prior to submission. **If any files are excluded, or fail to pass the checker, the grader *will* fail, resulting in a 0 on the assignment.**

Part 1 - Undirected Graphs

If you want to perform any further operations on graphs, you must first have a data structure to operate on. This section will concentrate on building such a structure.

To begin with, we will start with what we know: that graphs consist of objects, and their relationship to one another. More formally, that a graph $G = (V, E)$ where V is the set of vertices (things) and E is the set of edges (their relationships to one another).

Just building this into a computer, however, is not straight-forward. As covered in lecture, taking the object-oriented approach here exhibits bad locality

of reference. Therefore, we want to something a bit more sophisticated than our previous work with trees.

```
import java.util.*;

public interface IGraph<T> {
  public ArrayList<T> getVertices();
  public ArrayList<ArrayList<Float>> getEdges();

  public void addVertex(T v);
  public void addEdge(T v1, T v2, float w);

  public void removeVertex(T v);
  public void removeEdge(T v1, T v2);
}
```

The vertices should be placed into an array list in the order in which they were added. For example, if I added 4 vertices v_0, v_1, v_2, v_3 into an array, they should be at indices 0, 1, 2, and 3, respectively.

Each of the `addVertex` and `addEdge` calls should behave in the following manner. When a new (unique) vertex is added, that vertex is assigned a number (based on its index in the vertices array) that serves as an index into the edges table. Any new vertices added in this manner are completely disconnected from the rest of the graph by default. If an edge is added, it obviously makes no sense to add it between one or two objects that do not exist in the graph, in such cases, an error should be thrown. **The edges for a graph are expected to be represented as an upper-triangular matrix!**

Similarly, trying to delete a vertex or an edge between two vertices with one or more of them potentially not being part of the graph already, should result in no operation being performed. Only deletes on well-defined vertiecs and edges are to be accepted.

You should not return the internal data structures as the result of a `getVertices` or `getEdges` call! If you do this, it is possible for someone to change their values without you knowing. For example, if I were to remove a vertex from the list of vertices you had returned, there might still be a reference to it in the edges. Instead, you **must** clone the internal class members in order to ensure this does not happen.

The checker for this project part is as follows:

```
import java.util.*;

public class GraphCheck {
  public static void run () {
    IGraph<Integer> gInts = new Graph<Integer>();
    IGraph<String> gStrs = new Graph<String>();
    IGraph<Float> gFlts = new Graph<Float>();
```

```
    gInts.addVertex(1);
    gStrs.addVertex("test");
    gFlts.addVertex(0.5f);

    gInts.addVertex(2);
    gInts.addEdge(1, 2, 1.0f);

    gStrs.addVertex("another");
    gStrs.addEdge("test", "another", 1.0f);

    gFlts.addVertex(2.0f);
    gFlts.addEdge(0.5f, 2.0f, 1.0f);

    ArrayList<Integer> vInts = gInts.getVertices();
    ArrayList<String> vStrs = gStrs.getVertices();
    ArrayList<Float> vFlts = gFlts.getVertices();

    ArrayList<ArrayList<Float>> eInts = gInts.getEdges();
  }
}
```

Part 2 - Connected Components

A connected component is simple a subset of vertices in a graph that are all reachable from one another by following some path between them. A graph is considered connected if it contains one connected component – consisting of all the vertices in the graph. In this section, we will used a graph traversal to determine which connected component a vertex belongs in.

```
import java.util.*;

public interface IConnected<T> {
  ArrayList<T> componentOf(T v);
}
```

This single-function interface requires that you start at a given vertex, and find all other vertices in the graph connected to it. That is, that you return the connected component which includes the vertex passed to the function as an argument.

If a vertex is provided to the `connectedTo` function that does not exist in the graph, you must return an empty `ArrayList` to receive full credit for this part of the assignment.

A simple checker for this part follows:

```
import java.util.*;
```

```
public class ConnectedCheck {
  public static void run () {
    IConnected<Integer> cInts = new Graph<Integer>();
    IConnected<String> cStrs = new Graph<String>();
    IConnected<Float> cFlts = new Graph<Float>();

    ArrayList<Integer> ints = cInts.componentOf(1);
    ArrayList<String> strs = cStrs.componentOf("test");
    ArrayList<Float> flts = cFlts.componentOf(3.0f);
  }
}
```

Hint: You might want to consider implementing a private method, `adjacentTo`, which will return all vertices adjacent (one edge away from) the given vertex.