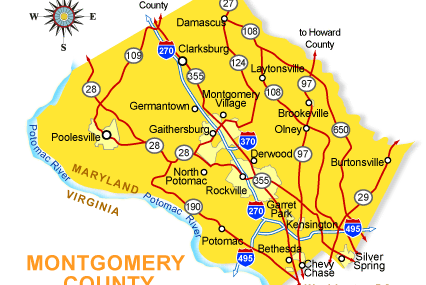
**CMSC204**

**Assignment #6**



Behind every Google Map, there is a much more complex structure that's the key to your queries but hidden from your view. It contains the logic of places: their no-left-turns and freeway on-ramps, speed limits and traffic conditions. This is the data that you're drawing from when you ask Google to navigate you from point A to point B.

**Assignment Description**

In this project you will be creating an application to maintain a network of towns and the roads connecting them. The application will use Dijkstra’s Shortest Path algorithm to find the shortest distance between any two towns. Upload the initial files and your working files to the repository in GitHub you created in Lab 1, in a directory named Assignment6.

**Concepts tested by this assignment**

Implement Graph Interface

Use Graph to maintain a network of Vertices

Implement Shortest Path Algorithm

**Classes**

**Data Element – Town (Vertex)**

Create a Town class that holds the name of the town and a list of adjacent towns, and other fields as desired, and the traditional methods (constructors, getters/setters, toString, etc.). It will implement the Comparable interface. This is the class header:

public class Town implements Comparable<Town>

Two towns will be considered the same if their name is the same.

**Data Element – Road (Edge)**

Create a class Road that can represent the edges of a Graph of Towns. The class must implement Comparable. The class stores references to the two vertices(Town endpoints), the distance between vertices, and a name, and the traditional methods (constructors, getters/setters, toString, etc.), and a compareTo, which compares two Road objects. Since this is a undirected graph, an edge from A to B is equal to an edge from B to A. This is the class header:

public class Road implements Comparable<Road>

**Data Structure – Graph, implements GraphInterface**

Create a Graph class that implements the GraphInterface given you. For Graph<V,E>, V is the vertex type (a Town), E is the edge type (a Road). You will need to decide how to store the graph, use an adjacent matrix or an adjacency list. This is the class header:

public class Graph implements GraphInterface<Town, Road>

Within the Graph interface is a method shortestPath, which finds the shortest path from a given source Town to a destination Town. Since there is a unique shortest path from every vertex to the source, there is a back-pointer to the previous vertex. The method shortestPath calls dijkstraShortestPath which finds the shortest path from the source to every other vertex in the graph. You will be coding the Dijkstra’s Shortest Path algorithm. You will then be able to find the connections between two towns through the roads that connect them.

You may use the adjacency matrix approach found in the text book, or you may use a set of Towns and a set of Roads. The ShortestPath algorithm typically uses a weighted graph which means that the edges have a weight, and this is used to determine the shortest path. For this implementation, each weight will be the distance of the road in miles.

**Data Manager – implements TownGraphManagerInterface**

Your TownGraphManager will hold an object of your Graph. Implement the TownGraphManagerInterface. There are methods to populate the graph (reading from a text file), add a town (vertices), add a road (edge), list all towns and all roads, and list towns adjacent to a given town.

Your solution will find the shortest path from a start town to a destination town. It will account for the possibility of a disjoint graph (i.e., not all vertices can be reached from all other vertices.)

You may add any methods as needed for your design.

**Exception Classes**

FileNotFoundException – created and thrown when the selected input file is not found.

IOException – created and thrown when user selects an input file that cannot be read (check out the methods of File).

Note that these exceptions exist in the Java API.

**GUI Driver (provided for you)**

The GUI has four sections: a Town section where you can add towns, a Road section where you add roads, a Find Connection section, and an administration section. It has a Read File button, which allows the text files provided to be read and populate the graph.

**Testing**

1. Your completed implementation must pass the TownGraphManagerTest.java and the GraphTest.java.
2. The tests marked GFA (“Good Faith Attempt”) are the minimal testing that must pass in order for your implementation to meet the good faith attempt.
3. Create a JUnit Test - TownGraphManager\_STUDENT\_Test. Test all the methods of the TownGraphManager with a different set of data than the TownGraphManagerTest provided for you.
4. Create a JUnit Test – Graph\_STUDENT\_Test. Test all the methods of the Graph with a different set of data than the GraphTest provided for you.
5. Create a Junit Test – Road\_STUDENT\_Test. Test all the methods of your Road class.
6. Create a Junit test – Town\_STUDENT\_Test. Test all the methods of your Town class.

**Assignment Details**

**Populating the Data Structure**

You will be reading from a data file. You are provided with two sample files: MD Towns.txt and US Towns.txt along with two PowerPoint slides showing these graphs.

The Towns.txt files hold the information for the individual Towns and Roads, and is in the following format:

road-name,miles;town-name; town-name

For example:

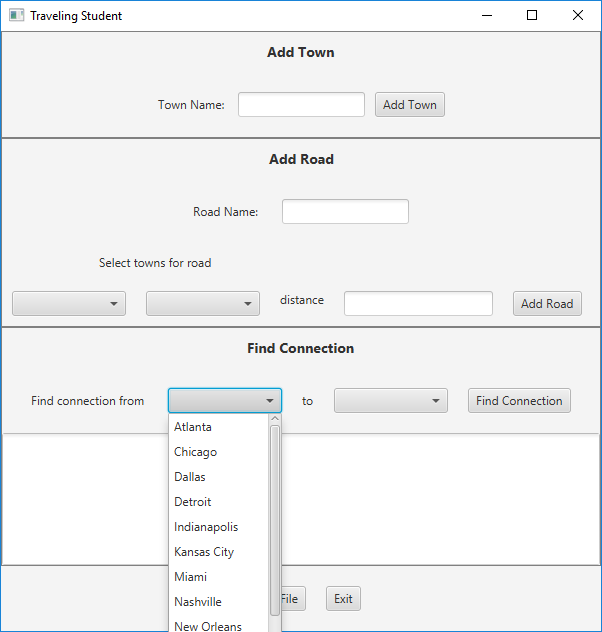
I-94,282;Chicago;Detroit

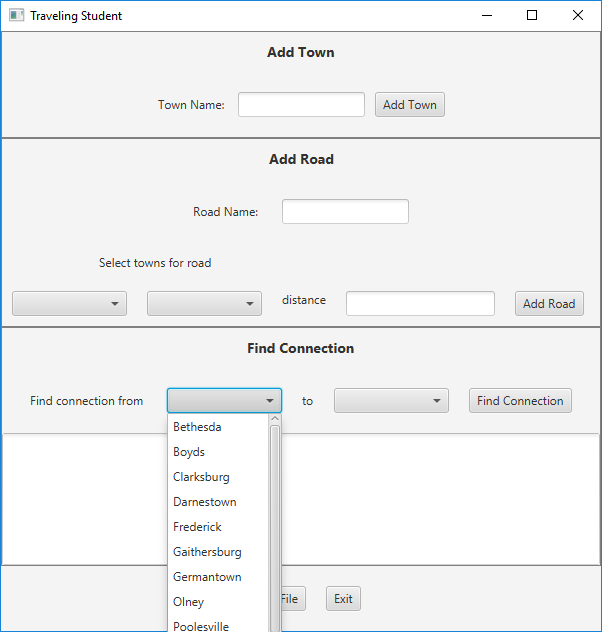
Notice that the road-name and miles are separated by a comma, while the road information and the two towns are separated by semi-colons.

After reading these files, you will have an initial set of vertices and edges in your Graph.

**Examples**

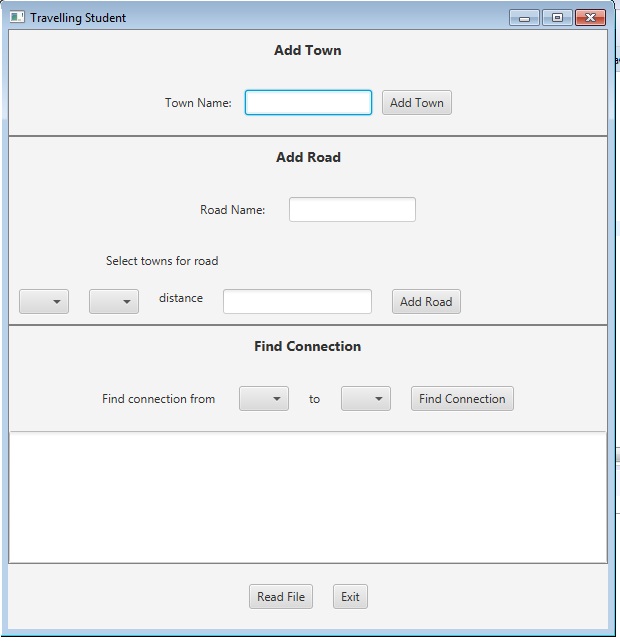
**After reading in the MD Town file After reading in the US Town file**





**The GUI (Provided for you)**

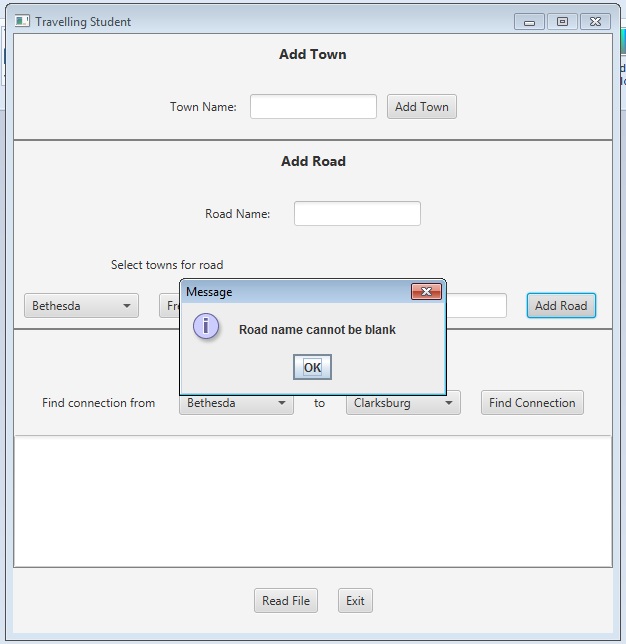
The GUI will have four sections: an Add Town section, an Add Road section, a Find Connection section, and an administration section. There will be four ComboBoxes each containing the same list of Towns. On startup the graph will be empty.



**Add a Town Button**

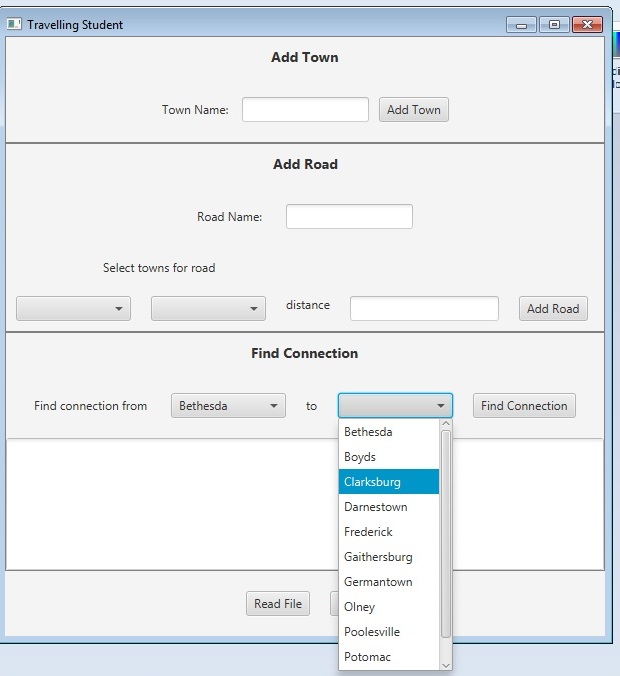
The user may add a new Town by typing its name in the textfield. If the textfield is blank when the Add Town button is selected, the GUI should show an error message. When a new Town is added, the TownGraphManager will add it to the graph, and the Town’s name will be added to the four ComboBoxes.

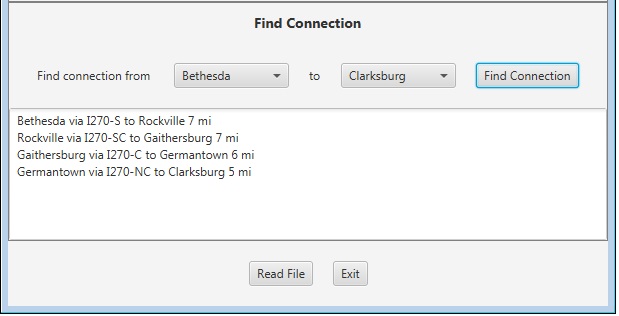
**Add a Road Button**

To add a road, a town must be selected from each of the two ComboBoxes in the Add Road section, an integer distance entered, and a road name entered. When the Add Road button is selected, the edge is created and entered in the graph.

**Find Connection Button**

Display all the available towns in the ComboBoxes (in alpha order by name). When the user selects the towns, display the name in the ComboBoxes. When the user selects the “Find Connection” button, the TownGraphManager’s shortestPath method is called. The resulting list of roads connecting towns, and the distance along each road, is displayed in the text area.





If the “source” town and “destination” town are the same, or if there is no route between the two, state that in the text area.

**Read File Button**

The Towns.txt files hold information for individual Towns and Roads, and is in the following format:

road-name**,**miles**;**town-name**;**town-name

For example:

I-94,282;Chicago;Detroit

Notice that the road-name and miles are separated by a comma, while the road information and the two towns are separated by semi-colons.

**Exit Button**

The program will terminate.

**Deliverables**

**Deliverables / Submissions:**

Design: UML class diagram with algorithm (pseudo-code) for methods

Implementation: Submit a compressed file containing the follow (see below): The Java application (it must compile and run correctly); Javadoc files in a directory; a write-up as specified below. Be sure to review the provided project rubric to understand project expectations. The write-up will include:

* UML diagram
* In three or more paragraphs, highlights of your learning experience

**Deliverable format:** The above deliverables will be packaged as follows. Two compressed files in the following formats:

* LastNameFirstName\_Assignment4\_Complete.zip, a compressed file in the zip format, with the following:
  + Write up (Word document) - reflection paragraphs
  + Final Design: UML Diagram - latest version (Word or jpg document)
  + Screen shot of GitHub repository with final set of files.
  + doc (directory) - Javadoc
    - * File1.html (example)
      * File2.html (example)
  + src (directory) 
    - * File1.java (example)
      * File2.java (example)
* LastNameFirstName\_Assignment4\_Moss.zip, a compressed file containing one or more Java files:
  + File1.java (example)
  + File2.java (example)

This folder should contain Java source files only