CSCI 2270 – Data Structures and Algorithms
Instructor: Hoenigman
Assignment 11_ExtraCredit
Due Wednesday April 13, by 3pm

This Extra Credit assignment is Part 2 of Assignment 10. It will count as an assignment in your final grade. If you missed an assignment, this can replace it. If you haven't missed an assignment, I will still add the points for this one to your assignments.

This assignment builds on Assignment 10 by including a shortest distance calculation between two cities using Dijkstra's algorithm, as well a depth-first search through a different data set to find the shortest path through 10 cities.

A Nation Divided … by Zombies!

Admit it, we all love zombies. Maybe it's because they don't actually exist, and we don't actually have to worry about navigating a life among the undead. But, imagine for a second an alternate universe where they do exist and they have attacked – creating mayhem throughout the country, knocking down communications towers and taking control of bridges and highways. One could imagine a resourceful zombie coalition making it impossible to travel between major cities, isolating human survivors in small districts around the country with no safe means of reaching other districts. The US would become a collection of small outposts, where cities within a district could be reached from within the district, and district residents would need to be careful about travel even within their district. Knowing the shortest path between cities to avoid being attacked would be paramount for survival.

What your program needs to do:

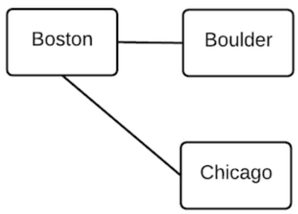
Build a graph. There is a file on Moodle called zombieCities.txt that contains the names of 15 cities and the distances between them stored as an adjacency matrix. Cities that still have roads connecting them that aren't controlled by zombies have a positive distance in the file. Cities that have been cutoff from each other have a -1 as their distance. When the user starts the program, read in the cities and distances from the text file and build a graph where each city is a vertex, and the adjacent cities are stored in an adjacency list for each vertex. For this assignment, you will not need to use the actual distance, only the fact that there is an edge connecting two cities. For the next assignment, you will need the distances, so it is probably best just to set that up now.

Use a command-line argument to handle the filename.

For example, this data:

	Boston	Boulder	Chicago
Boston	0	2000	982
Boulder	2000	0	-1
Chicago	982	-1	0

would generate this graph:



The vertices in the graph are Boston, Boulder, and Chicago. The adjacent vertices for Boston are Boulder and Chicago. The adjacent vertex for Boulder is Boston, and the adjacent vertex for Chicago is Boston.

Display a menu. Once the graph is built, your program should display a menu with the following options:

1. **Print vertices**
2. **Find districts**
3. **Find shortest path**
4. **Find shortest distance**
5. **Road trip**
6. **Quit**

Menu Items and their functionality:

1. **Print vertices.** If the user selects this option, the vertices and adjacent vertices should be displayed. The district ID should also be included in the display. (There is more about what the district ID is in the Find Districts menu item.)

   An example of how the output should be formatted is shown here:
   1:Boston->Boulder***Chicago
   1:Boulder->Boston
   1:Chicago->Boston

   The 1 shown is the district ID. District IDs should all be initialized to -1. If you call print vertices before finding districts, your display would look like:
   -1:Boston->Boulder***Chicago
   -1:Boulder->Boston
   -1:Chicago->Boston

2. **Find districts.** If the user selects this option, you need to do a breadth-first search of the graph to determine the connected cities in the graph, and assign

those cities the same district ID. The connected cities are the vertices that are connected, either directly by an edge, or indirectly through another vertex. For example, in the Boulder, Boston, Chicago graph shown above, these three cities are all connected even though there isn't an edge connecting Chicago and Boulder. There is a path between these two cities that goes through Boston.

In your graph, add a parameter to each vertex to store a district ID. The ID should be an integer, 1 to *n*, where *n* is the number of districts discovered in the graph, you will not know this value ahead of time. To get the correct, expected district ID for each vertex, make sure you read in the zombieCities.txt file in order so that your vertices are set up in alphabetical order.

When assigning district IDs, start at the first vertex and find all vertices connected to that vertex. This is district 1. Next, find the first vertex alphabetically that is not assigned to district 1. This vertex is the first member of district 2, and you can repeat the breadth-first search to find all vertices connected to this vertex. Repeat this process until all vertices have been assigned to a district.

You do not need to print anything for this menu option. To verify that district IDs have been assigned, call print vertices again.

3. **Find shortest path.** If the user selects this option, they should be prompted for the names of two cities. Your code should first determine if they are in the same district. If the cities are in different districts, print "No safe path between cities". If the cities are in the same district, run a breadth-first search that returns the number of edges to traverse along the shortest path, and the names of the vertices along the path. For example, to go from Boulder to Chicago in the example graph, you would print:

   2, Boulder, Boston, Chicago

   You will need to include a distance parameter in each vertex. There are multiple approaches to storing the path and you are welcome to use any approach that works for you. Some options include storing the parent for each vertex visited on the search path, or storing the path information in an array or queue and reconstruct the path from that data. There is more information about storing the path information in an array in your book.

4. **Find shortest distance.** If the user selects this option, they should be prompted for the names of two cities, just as in the shortest path menu option. If the cities are in different districts, print "No safe path between cities". If the cities are in the same district, run Dijkstra's algorithm to find the shortest weighted path between the cities. Print the distance between the

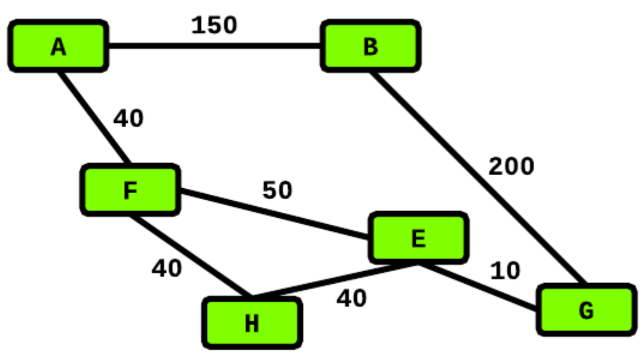
cities, and the names of the cities on the path between them. (Zombies like cities, so a longer, weighted path that avoids cities might be preferred to a shorter path that travels through more cities.) For example, to go from Boulder to Chicago in the example graph, assuming the edge weights were 50 each, you would print:

100, Boulder, Boston, Chicago

5. **Road trip.** Order has been restored, and it's time for a road trip. You have compiled a list of the 10 best places to visit in the Western US, given in bestPlaces.txt on Moodle. Leaving from Boulder, calculate the shortest distance to travel between all 10 places without visiting any place more than once and then return home to Boulder.

   The problem of determining the shortest path that includes all cities in the graph is slightly different than finding the shortest path between two cities, therefore you can't use Dijkstra's algorithm for this problem. The shortest path between two cities doesn't need to travel through all cities. The only way to know for sure that you have found the shortest distance through all cities is to calculate all possible paths and select the shortest one.

   For example, consider the graph shown here:



   You could use Dijkstra's algorithm to find the path between A and G with the shortest distance. However, for the road trip question, you need to find the path that includes all vertices, A, B, G, F, E, and H that has the shortest distance. This problem is not dissimilar to a previous recitation question where you were asked to find the maximum height of a branch in a binary search tree.

   The bestPlaces.txt file should be a command line argument in your code. Your program only needs to handle one file at a time. When you test your shortest distance code, use the zombieCities.txt file to build your graph and when you're testing the road trip code, use the bestPlaces.txt file to build your graph. There are many ways to solve this problem. Use can use a depth-first search, either recursive or iterative. You could also use a breadth-first search that generates a priority queue. Any algorithm that produces the correct answer will be accepted. Your program needs to print the distance travelled and the path, including the starting city at the beginning and ending of the path. For example, a path with a distance of 2000 starting in Boulder and travelling through three cities would look like:

2000, Boulder, Kansas City, Des Moines, Fort Collins, Boulder

How to submit your assignment
To submit your work, zip all files together and submit them to COG as Assignment11.zip.

Since this assignment is extra credit, we won't be doing grading interviews. Your code needs to work on COG to get credit.

Additional information
There is sample code on moodle showing how to use vectors and add vertices and edges to a graph. There is also code showing how to use the built-in C++ queue class that will make it easier to do the breadth-first search. The files are labeled **IntroToGraphs_C++Code.zip** and **QueueExample.zip**.

For more information on Vectors, there is a great tutorial here:

http://www.codeguru.com/cpp/cpp/cpp_mfc/stl/article.php/c4027/C-Tutorial-A-Beginners-Guide-to-stdvector-Part-1.htm

Appendix A – cout statements

1. Print menu

```
cout << "======Main Menu======" << endl;
cout << "1. Print vertices" << endl;
cout << "2. Find districts" << endl;
cout << "3. Find shortest path" << endl;
cout << "4. Find shortest distance" << endl;
cout << "5. Road trip" << endl;
cout << "6. Quit" << endl;
```

2. Print vertices

```
cout<< vertices[i].district
<<":"<<vertices[i].name<<"-->";
for each adjacent vertex:
       cout<<vertices[i].adj[j].v->name;
```

```
        if (j != vertices[i].adj.size()-1)
            cout<<"***";
```

3. Find districts

   Nothing to print.

4. Find shortest path

   ```
   cout << "Enter a starting city:" << endl;
   cout << "Enter an ending city:" << endl;
   ```

   One or both cities not found:
   ```
   cout << "One or more cities doesn't exist" << endl;
   ```

   Cities in different districts:
   ```
   cout << "No safe path between cities" << endl;
   ```

   Districts not set yet:
   ```
   cout << "Please identify the districts before checking
   distances" << endl;
   ```

   Algorithm successful:
   ```
   cout << edgesTraversed;
   for all cities in path
       cout << "," << path[j]->name;
   cout << endl;
   ```

5. Find shortest distance

   Same as shortest path.

6. Road trip

   ```
   cout << "Enter a starting city:" << endl;
   cout << distance;
   for all cities in path
       cout << "," << path[j]->name;
   cout << endl;
   ```

7. Quit

   ```
   cout << "Goodbye!" << endl;
   ```