CSCI 2270 – Data Structures and Algorithms

Instructor: Godley

Due Friday, July 27th by 11:59pm

Zombies – Part II

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:

In Part I of this assignment, you were asked to read data from *zombieCities.txt* and build a graph. In this assignment, you will enhance the graph you created in Part I and find connected vertices, sub-graphs, and the shortest path between vertices.

Build a graph. There is a file on Moodle called *zombieCities.txt* that contains the names of 10 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.

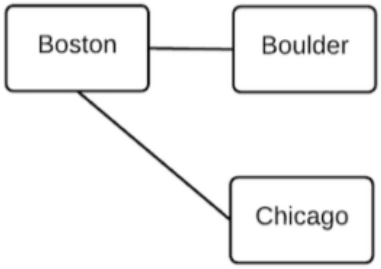
Use a command-line argument to handle the filename.

Your program needs to open the *zombieCities.txt* file using command-line arguments. When we test your code, we will use a file with a different name.

In the adjacency matrix in the text file, you might see data that looks like this:

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

That matrix 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. **Fined Shortest Weighted Path**
5. **Quit**

Menu Items and their functionality:

1. **Print vertices**. If the user selects this option, the vertices and adjacent vertices should be displayed. The vertices all have a property called districtID to identify which sub-graph they belong to. 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 depth-first search (DFS) 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 the .hpp file provided, the graph, has a districtID parameter for each vertex. The ID is 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 depth-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 (BFS) 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

   There is a distance parameter and a parent parameter in each vertex. Reconstructing the path is easiest if you store the parent for each vertex visited on the search path, and then traverse back to the starting vertex using the parent.

4. **Find shortest weighted 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 Dijkstra's algorithm to find the shortest distance between the two cities. You need to print both the distance and the names of the vertices along the path.

Each vertex in the graph also includes a weighted distance and parent parameter. Reconstructing the path is easiest if you store the parent for each vertex visited on the search path, and then traverse back to the starting vertex using the parent.

Additional information

There is sample code on moodle showing how to use vectors and add vertices and edges to a graph. Feel free to use that code as the starting point for this assignment. Do not modify the *Graph.h* file. 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 weighted path" << endl;
cout << "5. 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 and shortest weighted 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 << minDistance;
for all cities in path
        cout << "," << path[j]->name;
```

```
cout << endl;
```

5. **Quit**

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