CS20 – Section 093 – Fall 2019

Programming Assignment 10 – Due Sunday December 15 at 11:59PM

Graphs (60 points)

In this assignment, you'll:

read a set of data representing a directed, unweighted graph
build an in-memory graph structure using the data
display the graph using depth-first traversal
display the graph using breadth-first traversal

1. Input data

The data consists of records like this:

16 3 15 4 -1

This represents a vertex 16 with neighbors 3, 15, and 4. The -1 is the indicator that no more neighbors for this vertex exist. A vertex with no edges would be represented as follows:

12 -1

Vertex 12 has no edges.

2. Graph data structure

The graph data structure is an adjacency list using STL vectors as the underlying structure. It's based on this struct which represents a vertex:

```
struct  Vertex
        int vertexValue;
        bool visited;
        vector <int> neighbors;
```

Each vertex has some integer value. There is no relationship between a vertex's value and its position in the vector; the vertex element 6 might be for the vertex with value 13. Each vertex has some number of neighbors. You don't know in advance how many vertices will be in the graph, and you don't know how many neighbors a vertex will have, so it makes sense to have a data structure which is completely flexible in both dimensions. See the "vector of vectors" cheat sheet for examples of how this is coded in C++.

3. Build and populate the graph

Main will instantiate the graph. Code a function named buildGraph, which is passed the graph:

```
void buildGraph (vector <Vertex> name)
```

buildGraph opens the data file. If the open fails, display an error message and exit. For each data record, one vertex is created. You may assume that the data is valid and that each value is separated by spaces. (This means you can use the >> operator). You may also assume that the graph contains at least one vertex.

Before returning, buildGraph should display the graph to make sure that it has been built correctly. See the sample output at the end for an example. It is highly recommended that you code and test buildGraph thoroughly before continuing.

4. Vertex lookup function

buildGraph builds a data structure that contains vertices, each with a unique value. As stated earlier, there is no relationship between the value of a vertex and its position in the data structure. Therefore, you cannot use the vertex value as the index (subscript) into the graph data structure. Although this assignment uses integer values, the values could as easily be strings, or even other data structures. You will therefore need a table lookup function, which, given a vertex value, returns the index into the vector for the vertex with that value.

5. Traverse the graph using depth-first traversal

After building the graph, main calls function dft to traverse the graph:

```
void dft (vector <Vertex> name)
```

dft traverses the graph starting with the first vertex. The first vertex is defined as the first vertex in the graph vector. dft loops through the vertices and calls dftInternal repeatedly with a new vertex each time. dftInternal, in turn, processes all the neighbors for that vertex.

dft is actually a helper function for dftInternal, which is recursive:

```
void dftInternal (vector <Vertex> name, vertex)
```

When called, dftInternal processes all the neighbors for whichever vertex which has been passed as a parameter. It loops through all its neighbors. A given neighbor might have already been visited; in this case, dftInternal goes on to the next neighbor. Otherwise, it calls itself when it finds a neighbor that has not already been visited.

Refer to the class discussion and the logic to understand how to code dft and dftInternal.

6. Traverse the graph using breadth-first traversal

The function bft traverses the graph using the breadth-first method:

```
void bft (vector <Vertex> name)
```

Like dft, bft starts with the first vertex, which is defined as the first vertex in the graph vector. bft is iterative (as opposed to recursive) and uses a queue to keep track of which vertices to process.

bst loops through all the vertices, pushing a vertex onto the queue when it finds one that has not yet been visited. Within that loop, it empties the queue, processing each vertex by pushing any neighbors that have not been visited into the queue. In this manner, all the vertices are pushed onto the queue in breadth-first order. As the queue is emptied, the neighbors for each vertex are examined to determine if they have been visited. If not, they are added to the queue and the process continues.

Use your queue class from assignment 6 as the queue implementation.

Refer to the class discussion and the logic to understand how to code bft.

7. Main

Code a main which instantiates the graph data structure. Main then calls buildGraph, dft, and bft, and exits. The output should resemble something like this (if using the graph from Malik p.696):

```
Populated graph:

vertex number 0 value 0 neighbors-> 1 5
vertex number 1 value 1 neighbors-> 2 3 5
vertex number 2 value 2 neighbors-> 4
vertex number 3 value 3 neighbors->
vertex number 4 value 4 neighbors-> 3
vertex number 5 value 5 neighbors-> 6
vertex number 6 value 6 neighbors-> 8
vertex number 7 value 7 neighbors-> 8 3
vertex number 8 value 8 neighbors-> 10
vertex number 9 value 9 neighbors-> 4 7 10
vertex number 10 value 10 neighbors->

Depth-first traversal:

0 1 2 4 3 5 6 8 10 7 9

Breadth-first traversal:

0 1 5 2 3 6 4 8 10 7 9

Have a great day!
```