CMPSCI 187 (Fall 2018) Lab 12: Graph Search

The goal of this lab is to get familiar with graph search techniques, by implementing four methods (make a complete graph, compute vertex valence, perform DFS from a given vertex, and find a path between two vertices using BFS). To work on the assignment:

- Go to ***File -> Make a Copy*** to make an editable copy of this Google Doc for your account
- Follow the instructions to complete the assignment
- When you are done, go to ***File -> Download As -> PDF Document***
- Log in to Gradescope and submit your PDF

The lab is due by 5:00 pm today.

Section 1: Make a Complete Graph

Import the starter code to Eclipse, and take a look at `Graph.java`. It contains the definition of a `Vertex` class, and a `Graph` class. A graph is made of a list of vertices, and an adjacency matrix. For this project, we focus on unweighted graph, so the adjacency matrix is of `boolean` type -- a `true` value means there exists an edge from vertex `i` to `j`, and a `false` value means the edge does not exist. Note that the graph may be either undirected or directed, which means the matrix may be symmetric or may not.

For the first problem, you need to write a method that makes a complete graph, that is, a graph where every two vertices are connected by an edge (but there is no a self-loop). You do NOT need to modify the number of vertices, instead, modify the adjacency matrix. **HINT**: If you are having trouble figuring out how to do this, take a look at the lecture slides, and check what a complete graph looks like. Then reason about what the adjacency matrix should look like. Then write down the code to accomplish this task.

The file `GraphTest.java` contains one test for this method. After completing the method, run `GraphTest.java` and see if the printout of the adjacency matrix matches the one you have in mind.

```
public void completeGraph() { // 3 points

       for (int i = 0; i < adjMat.length; i++)

           for (int j = 0; j < adjMat.length; j++)

               adjMat[i][j] = i == j ? false : true;

}
```

Section 2: Find the Valence of a Vertex

Given a vertex id, you need to return the number of neighbors of the given vertex. That is, find out how many vertices that vertex is connected to. Think about how to compute this from the adjacency matrix, then write the code to accomplish this task. Return `-1` if the vertex id is out of bounds (i.e. less than 0 or larger than the largest vertex index).

```
/* Return vertex vid's valence. Return -1 if vid is out of bounds */
public int valence(int vid) {  // 2 points
	if (vid < 0 || vid > adjMat.length)
		return -1;
	int valence = 0;
	for (boolean isNeighbor : adjMat[vid])
		if (isNeighbor)
			valence++;
	return valence;
}
```

Section 3: Depth First Search (DFS)

Now you will implement depth first search (DFS) for graph traversal. Given a starting vertex, your task is to print out every vertex visited during DFS. Review lecture slides about DFS. In particular, the code for DFS is largely provided to you in the slides. Note that each vertex has a variable called `visited` that you can use to mark a vertex as visited during DFS. Also, a stack has been created for you below. Pay attention to the format in which you are asked to print out the vertices (i.e. create a space between the printouts of every two vertices). Run `GraphTest.java` to see if your result matches the solution.

```
public void DFS(int start) {  // 5 points
	for (int i = 0; i < numVerts(); i++)
	    verts.get(i).visited = false; // clears flags
	Stack<Integer> stack = new Stack<Integer>(); // create stack
	stack.push(start);
	verts.get(start).visited = true;
	System.out.print(verts.get(start) + " ");

	while (!stack.isEmpty()) {
	    int index = stack.peek();
	    int neighbor = -1;
	    for (int i = 0; i < adjMat.length; i++) {
	        if (adjMat[index][i] && !verts.get(i).visited) {
	            neighbor = i;
	            break;
	        }
	    }
	    if (neighbor == -1)
	        stack.pop();
	    else {
	        stack.push(neighbor);
	        verts.get(neighbor).visited = true;
	        System.out.print(verts.get(neighbor) + " ");
	    }
	}
}
```

Section 4: Pathfinding using Breadth First Search (BFS)

You will implement BFS to find a path between two vertices in a graph. Given a `start` index and an `end` index, this method finds a path between the two vertices using BFS, and returns it in the form of a Java `ArrayList`. The first element in the `ArrayList` must be **`start`** (i.e. the start index); the last element must be **`end`**; and the intermediate vertices must be the indices of vertices on the path from start to end. Return null if there is no path.

HINT: review lecture slides of BFS code. Below the queue needed for BFS is already created for you. Note that the lecture slides did not contain code for BFS pathfinding, but does contain DFS pathfinding code, and discussions about how to implement BFS pathfinding. The part that deserves some thinking is how to reconstruct the BFS path once it's found. Note that you cannot simply add all vertices you visited during BFS to the ArrayList -- many of them may not be on the path from start to end. What kind of bookkeeping code and data structure do you need? A tip: you are allowed to add additional data members (e.g. a variable named `parent` or `predecessor`) to the `Vertex` class, to help you with the bookkeeping. Also note that the returned ArrayList must contain vertices in the correct order (not reverse order).

```
public ArrayList<Integer> findPathBFS(int start, int end) {  // 8 points
       for (int i = 0; i < numVerts(); i++)
           verts.get(i).visited = false; // clear flags
       Queue<Integer> queue = new LinkedList<Integer>(); // create queue
       queue.add(start);
       verts.get(start).visited = true;

       while (!queue.isEmpty() && queue.peek() != end) {
           int index = queue.remove();
           for (int i = 0; i < adjMat.length; i++) {
               if (adjMat[index][i] && !verts.get(i).visited) {
                      queue.add(i);
                      verts.get(i).visited = true;
                      verts.get(i).prevIndex = index;
               }
           }
       }
       if (queue.isEmpty())
           return null;
       ArrayList<Integer> path = new ArrayList<Integer>();
       for (int index = end; index != -1; index = verts.get(index).prevIndex)
           path.add(0, index);
       return path;
}
```