Lab 11: Graphs and Graph Algorithms

1. **Graph Representation**

   We can represent graphs in a number of different ways. The standard way is to use what are called adjacency lists. This means that each vertex in a graph has a list of vertices it is adjacent to. A graph is then just a collection of such vertices.

   In Java, we can then represent a graph as a LinkedList of vertices, with each vertex having a LinkedList of edges. An edge is just a pair of two vertices.

2. **Graph Search**

   One of the most common things to do with a graph is traverse it, that is visit all (or many) of the nodes. We will look at two different kinds of traversals of a graph. We will implement them in the context of finding a particular value in the graph, where values are attached to vertices.

   (a) **Depth First Search (DFS)**

   This is the simplest way of traversing a graph. We begin at a start node. We check if this node is the one we are looking for. If it is, we stop. Otherwise, we recursively search all the neighbors of the node. We can get all the neighbors easily, since they are in a list attached to the node.

   This simple algorithm works in some cases. However, in other cases it will run forever. Why is this? What kinds of graphs will cause this behavior?

   To solve this problem, we introduce a mark field on the vertices. After a node has been seen, it is marked. We never revisit a marked node.

   Implement this algorithm recursively.

   (b) **Breadth First Search (BFS)**

   Imagine that we are traversing a graph with one very long branch, and a number of short branches. If what we are looking for is in a short branch, DFS may still take a long time. How can we fix this?

The solution is BFS. In BFS, we visit all of the neighbors of a node before we descend further into the tree. This means we visit all the nodes one step away from the start, then two steps away, and so on.

However, this requires a more complicated algorithm. We need to keep a list of nodes to visit, and add new nodes on at the end of the list as we discover them. This requires using a new data structure to keep track of these nodes.

Implement this algorithm using the proper data structure.

(c) **DFS Again**

In languages like Java, lots of recursive calls can be bad for performance on a large graph. Fortunately, we can implement DFS using a while loop and a data structure very much like BFS.

Implement this algorithm.

What is the difference in how we use the data structure? How similar is the new DFS code to the old BFS code? *[Hint: DFS can be written with only 2 extra characters compared to BFS.]*

3. **Extending our graph representation**

So far, we have only seen unweighted graphs. However, in real life edges are not always the same. For example, in a road network, different routes take different amounts of time. Change our representation of edges to add weights. Then calculate the total weight of the edges you traversed in the BFS and the two DFS searches. Is one better than the other?

4. **Changing our graph representation**

Change the representation of graphs to be a list of edges instead of a list of vertices. Does this require you to change your DFS and BFS code?