**CS361 Final Project (Fall 2021)**

Fill out each section of this document. For a written assignment, feel free to handwrite the solutions if that is more convenient. For a programming assignment, the solution should be either a Java or Python program.

You should submit this document via Canvas no later than 11:59pm on Friday 12/17. If you have additional files to upload (e.g. Java or Python files), then please submit a **zipped** **directory** with all necessary files.

**Name: Tyler Weir**

**Algorithmic Topic: Graph Traversal Algorithms**

**Homework Instructions:**

Please type the instructions for your homework assignment. Give clear and detailed instructions as to what a good solution looks like. For example, does the student’s solution need to be written in a specific format? Do you want the worst-case complexity of their solution? For a Java program, does the structure of their code matter? Be specific.

### Description:

In this assignment students will use Java to implement three graph traversal algorithms (Depth First Search, Breadth First Search, and Dijkstra’s). The starter code contains a graphical implementation allowing users to visualize the algorithms as they explore a grid shaped graph.

### Instructions:

1. The starter code contains 3 packages: **graph, util**, and **visualizer**. Your work will start in the **graph** package. Here, add your Graph.java and GraphIfc.java files. Your Graph class must be able to handle generics. If you cannot create and add vertices of any specific type to a Graph instance with that specific type, then correct your program to be able to do so.
2. Next, you’ll work in the **util** package. Here, you should add Pair.java and your PriorityQueue.java files. Once added, your next task is to modify PriorityQueue to take generic elements and assign them priorities. This should not take too much work. You’ll start by changing the class header to public class PriorityQueue<V>. the field types will become Map<V, Integer> and List<Pair<Integer, V>>. The order of the generic and the integer may vary depending on your specific implementation. After that, go through and update the public methods and each of the necessary private methods to handle the new generic type definition. This should only require small changes in each method.
3. Lastly, you will work in the visualizer class. This is where the fun will start. Begin by running the main method of PathFindingVisualizer.java to get an understanding of what you’re going to do. You should see a small window pop up along with a message in the terminal. The window will contain a grid of white squares with one green square. The grid is a graphical representation of an undirected graph. Each square is a vertex (or node). Each square is connected to its adjacent squares. A white square means the square is active and able to be explored. You may click on squares to deactivate them (ie remove the vertex from the graph.) and they will turn black. The green square represents the starting node of the graph. You may change the position of the starting node by right clicking and active square. As your graph traversal algorithms explore the graph, they will mark squares as visited in which case they will turn red.

Take a look into the Node class under the util package. Each square in the grid is represented as an instance of Node. Take special note of the isVisited() method. This is the ONLY Node method you should use. Your algorithms will traverse a Graph<Node> graph. and will start at Node s.

Back in the PathFindingVisualizer.java file you should see three empty functions. These functions are dfs, bfs, and dijkstra. Using pseudocode from lecture, you must implement each of these three algorithms. Note that functions are void and should not return anything.

This part is important! **Nodes must be marked as visited using the visitNode(Node n) function.** This function calls n.visit() as well as repaint() and a small execution delay. This is essential for the graphical output of the algorithm to work. If you just use Node.vist() then the algorithm will have no graphical output and execution will be too fast to appreciate. Here’s some helpful tips for algorithm implementation: DFS can use Java’s built in Stack implementation Stack<Node>. BFS can use Java’s built in Queue<Node>. Note that Queue is an interface, so I used an instance of Java’s LinkedList<Node> as my queue. Finally Dijkstra’s should use your PriorityQueue<Node>. For the dist and prev arrays, you’ll have better luck using Map<Node, Integer> for dist, and Map<Node, Node> for prev.

After implementing the algorithms, run the main method again to see their graphical representations as they explore the graph. Play with moving the starting node and removing nodes to see how they affect the algorithms. Depending on your Graph implementation you may notice DFS follows a seemingly random path. For instance, my Graph uses a HashSet to store adjacent nodes which means the order they were added will not be reflected in the order they come out.

**Solution:** See code solution.

**Grading Rubric:**

Your homework assignment should be worth a total of 50 points.

Grading breakdown is as follows:

[10] Modified Priority Queue to handle generics.

[10] Working implementation of Depth First Search.

[10] Working implementation of Breadth First Search.

[10] Working implementation of Dijkstra’s Algorithm.

[5] Appropriate running times (Code is efficient)

[5] Code style