Shortest Path CS3 Lab Reflection

By: Vyom Manot

A key component of computer science is the way data is stored and how to access the stored data. In the class, "Computer Science 3 Advanced", I have been tasked to complete a lab called, "Shortest Path." This lab implements Dijstra's algorithm in order to traverse a graph with given edges (and their weights), the coordinate points of each vertex, and the desired route (shortest path) from one vertex to another. The objective of this lab is to find the shortest path (least-costly with respect to weight) to the desired vertex.

First, it is important to discuss how Dijstra's algorithm works. In essence, the algorithm takes a series of vertices, series of edge/connections between vertices, and a **Source Vertex** and an **End Vertex**. From there, we do a modified Breathd-First Search that proritizes distances from a specific vertex that is the shortest. Initially, all Vertex distances (except the **Source** which has distance set to 0) are set to be positive infinity. From here, we visit all vertices that neighbor our **Source**. We can name our currently visited Vertex (**Source** in this case), "**Current**." For each of the neighbors, we check the distance from **Current** to the neighbor, and if the distance is less than the set distance (which it will for the first iteration because initially distances are set to infinity), we set the distance of **Current's** neighbor to the new distance we just calculated. After this we will mark **Current** as visited and continue this same process with a new **Current** node (which is picked as the Vertex with the shortest distance attribute). This can be a lot to unpack so from a step-by-step perspective this is what it would look like:

1. Make all Vertices have a distance equal to infinity except Source Vertex
2. Current = Source (initally) or Current = Vertex with smallest distance (after first iteration)

3. For each neighbor that Current has
    a. Check to see if neighbor's distance from Current is less than its already set distance
        i. If it is, add Current's distance + distance from Current to neighbor and set neighbor's new distance to it.
4. Mark Current as visited and continue from step 2

There are many different ways of representing what a Vertex object should contain. The following image shows my implementation of the Vertex class.

```
import java.util.LinkedList;

public class Vertex implements Comparable<Vertex> {

    private int x;
    private int y;
    private int id;
    private boolean visited; // whether this vertex has been visited or not
    private double distance = Double.POSITIVE_INFINITY; // distance from source node

    private LinkedList<Vertex> adjVertices; // neighboring vertices
    private Vertex previousVertex;

    Vertex(int id, int x, int y) {

        this.id = id;
        this.x = x;
        this.y = y;
        adjVertices = new LinkedList<>();

    }
```

Here you can see that in my implementation, a Vertex contains its (x,y) position, it's ID, whether the Vertex has been visited or not, the distance (set to infinity initially), a LinkedList that keeps track of all the Vertices' neighbors, and a Vertex that keeps track of the previous vertex (EX: Current's neighbor's previous vertex *is* Current).

The hardest part for me deciding how do I want to represent my graph. Whether this be a AdjacencyMap, AdjacencyList, a Graph class, etc. I ended up choosing to make a AdjacencyList (array of vertices) for this project since that seemed to be a very simple and efficient design. Because of the way I made my Vertex class (as seen in the image), the vertices themselves are essentially LinkedLists and as such I only had to make an array of vertices. Another implementation that I considered and could have also worked was a Map application. This would be a map of key-value pairs where the values are lists of adjacent vertices linked to a specific vertex key.

Another part of this project was setting up the right variables to execute Dijstra's Algorithm. This would include taking in a Scanner input from a textfile that provides information on the number of vertices, the number of edges/connections, the position of the vertices, and the actual vertices we want to find the shortest path for.

```
Scanner sc = new Scanner(new File("H:\\ShortestPathCS3\\src\\input6.txt"));

int numOfVertex = sc.nextInt();
int numOfEdges = sc.nextInt();

Vertex[] arr = new Vertex[numOfVertex]; // A vertex is essentially a LinkedList. This would make this an AdjacencyList

for (int i = 0; i < numOfVertex; i++) {

    int id = sc.nextInt();
    int xPos = sc.nextInt();
    int yPos = sc.nextInt();

    arr[i] = new Vertex(id, xPos, yPos); // Filling up AdjList (arr) with all the vertices given in the problem


}

for (int i = 0; i < numOfEdges; i++) {

    int vertex1 = sc.nextInt();
    int vertex2 = sc.nextInt();

    arr[vertex1].getAdjVertices().add(arr[vertex2]); // vertex1 will create a connection to vertex2

}

int sourceIndex = sc.nextInt();
Vertex sourceVertex = arr[sourceIndex];
sourceVertex.setDistance(0); // sets the source node to have a distance of 0 as per Dijstra's Algorithm

// by this point all vertex contain a linkedList that has all neighboring verticies that can be visited- set up for problem is done
```

The following code implementation above shows how I used the text file in order to fill my **array of vertices** (first for-loop) and added each connection (neighbors) between the Vertices (second for-loop).

The lab I followed showed a very different way to how to approach this problem. Frankly, the reason I didn't follow the lab's instructions is because it seemed excessive for the problem given at hand.The approach I took was a lot more simplistic.

Now we can finally get into how I actually implemented Dijsta's Algorithm into this problem.

```
int sourceIndex = sc.nextInt();
Vertex sourceVertex = arr[sourceIndex];
sourceVertex.setDistance(0); // sets the source node to have a distance of 0 as per Dijstra's Algorithm

// by this point all vertex contain a linkedList that has all neighboring vertiicies that can be visited- set up for problem is done


// Dijstra's Algorithm:
PriorityQueue<Vertex> q = new PriorityQueue<>();
int numberVisited = 0; // way of keeping track of how many Vertices we have visited- if this number reaches the length of the AdjList,
                       //  we will have visited all Vertices and can stop our algorithm

q.offer(sourceVertex); // offers the Source Vertex
while(numberVisited != arr.length) {
    Vertex current = q.poll();
(current != null) {
if (!current.isVisited()) {
    for (Vertex v : current.getAdjVertices()) {
        if (!v.isVisited()) {
        double distanceBetween = current.getDistance() + distance(current, v);
        if (distanceBetween < v.getDistance()) {
            v.setDistance(distanceBetween); // if new distance is shorter than old, replace it with new- this will reflect shortest path
            v.setPreviousVertex(current); // essentially a backwards LinkedList which keeps track of the shortest path from source to end
        }
        q.offer(v);
        }
    }
    current.setVisited(true);
    numberVisited++;
        }
    }
}

int endIndex = sc.nextInt();
Vertex endVertex = arr[endIndex];
System.out.println("Shortest distance from source vertex " + sourceVertex.getId() + " to end vertex " + endVertex.getId() + " is: "
        + endVertex.getDistance());
```

It follows the same step-by-step procedure described earlier in this document. The getDistance() method was simple a static method within the Main class that found the Euclidian Distance (distance formula) between two vertices using their respective (x,y) positions. Surprisingly, I didn't run into many issues with my code when writing this out. Here are a couple of the main ones (and how I fixed them) I did run into:

1. My PriotityQueue was choosing the Vertex with the longer distance
    a. Figured out quickly, it was because my overridden compareTo() method in my Vertex class was wrong- I had swapped my greater than and less than statements that decided priority.
2. Didn't realize I also needed to print the vertex path, not just the distance
    a. Realized I could add a **Vertex previousVertex** variable to my Vertex class and work backwards from the **End** vertex to the **Source** (like a backwards LinkedList) Then, starting from **End**, I printed each reoccurring **previousVertex.**
3. Just when I thought I was finished- I got a null error when testing with another text file/test case
    a. It was a quick simple fix where I added a null checker within my algorithm (Shown in image above)

Overall, this lab wasn't too hard. The longest and hardest part was deciding what kind of implementation of how I wanted to represent my graph/figuring out how I could represent my graph. I learned a lot about graphs and gained a deep understanding of how Dijstra's algorithm works. I was also intrigued by this kind of algorithm and I'm going to continue to learn similar kinds like the A* Algorithm, which includes heuristics to make the algorithm faster in general cases.