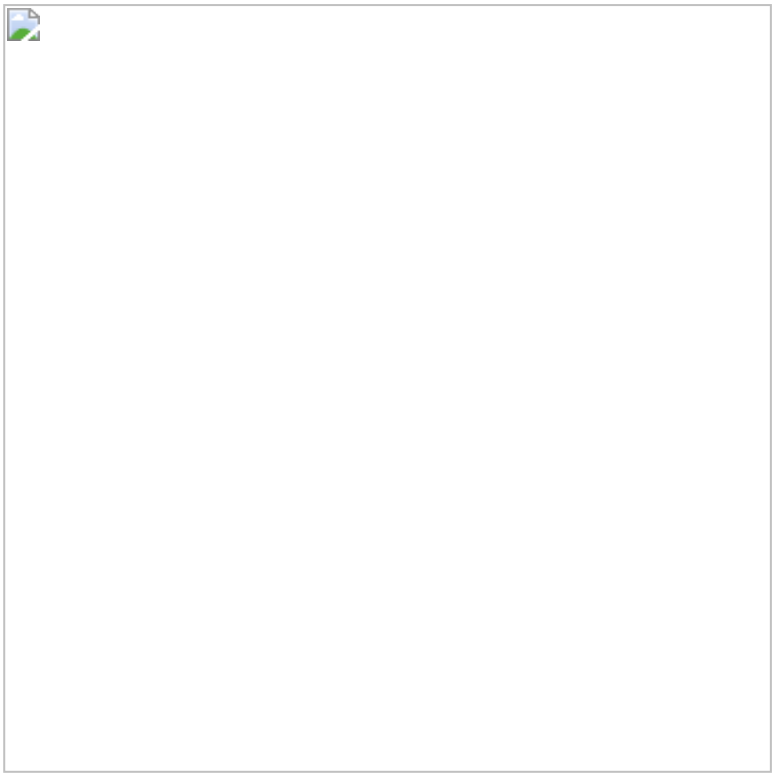
Project 9: Graph

Abstract

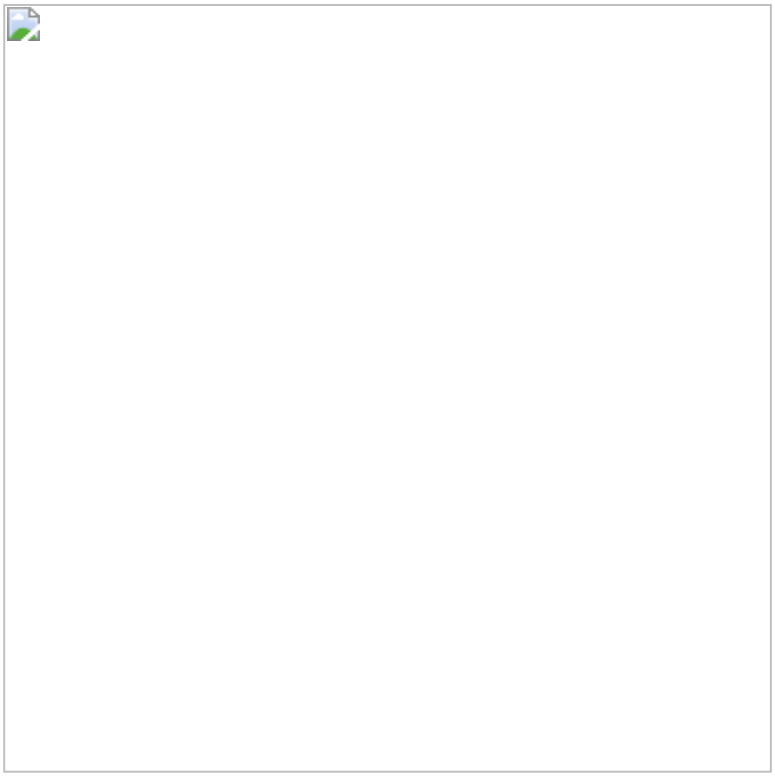
In this week's project, we implemented the data structure of Graph. In a graph, each data is stored as a vertex, and each vertex can have edges connecting to another edges. The edge can be uni-edge meaning uni-directional edge from one vertex to another, or bi-edge, meaning bi-directional edge between two vertices. I created a Vertex class, used an ArrayList to store all the connections that leave from the vertex. I also implemented Dijkstra's algorithm to compute the shortest path from one vertex to all other vertices in a graph. I tested my shortestPath method, and it worked correctly.

Result

In my main function of Graph class, I constructed 10 vertices, all of which are given different x and y coordinates. Then I set two vertex as the root and call the shortestPath method on them respectively, to see the cost of traveling to all other vertices from that root. Before setting a root, the graph looks like the following.



After setting Vertex at (0,0) as root, the cost of all vertices is written on the vertex.



After setting Vertex at (2,1) as root, the cost of all vertices after the shortestPath method is as follows.

Acknowledgement

I did the project independently and didn't implement the game.

Notes

Setting Vertex at (0,0) as root, it output the following in the terminal.

```
>> Set Root (0,0)
>> After Dijkstra's
  -> Position: (0,0)
  -> Number of neighbors: 2
  -> Cost: 0.0
  -> Visited: true

  -> Position: (0,1)
  -> Number of neighbors: 3
  -> Cost: 1.0
  -> Visited: true

  -> Position: (0,2)
  -> Number of neighbors: 3
  -> Cost: 2.0
  -> Visited: true

  -> Position: (0,3)
  -> Number of neighbors: 1
  -> Cost: 3.0
  -> Visited: true

  -> Position: (1,0)
  -> Number of neighbors: 3
  -> Cost: 1.0
  -> Visited: true

  -> Position: (1,1)
  -> Number of neighbors: 4
  -> Cost: 2.0
  -> Visited: true

  -> Position: (1,2)
  -> Number of neighbors: 2
  -> Cost: 3.0
  -> Visited: true

  -> Position: (2,0)
```

```
  -> Number of neighbors: 3
  -> Cost: 2.0
  -> Visited: true

  -> Position: (2,1)
  -> Number of neighbors: 2
  -> Cost: 3.0
  -> Visited: true

  -> Position: (3,0)
  -> Number of neighbors: 1
  -> Cost: 3.0
  -> Visited: true
```

For Vertex at (2,1):

```
>> Set Root (2,1)
>> After Dijkstra's
  -> Position: (0,0)
  -> Number of neighbors: 2
  -> Cost: 3.0
  -> Visited: true

  -> Position: (0,1)
  -> Number of neighbors: 3
  -> Cost: 2.0
  -> Visited: true

  -> Position: (0,2)
  -> Number of neighbors: 3
  -> Cost: 3.0
  -> Visited: true

  -> Position: (0,3)
  -> Number of neighbors: 1
  -> Cost: 4.0
  -> Visited: true

  -> Position: (1,0)
  -> Number of neighbors: 3
  -> Cost: 2.0
  -> Visited: true

  -> Position: (1,1)
  -> Number of neighbors: 4
  -> Cost: 1.0
  -> Visited: true

  -> Position: (1,2)
  -> Number of neighbors: 2
  -> Cost: 2.0
  -> Visited: true

  -> Position: (2,0)
  -> Number of neighbors: 3
```

```
   -> Cost: 1.0
   -> Visited: true

   -> Position: (2,1)
   -> Number of neighbors: 2
   -> Cost: 0.0
   -> Visited: true

   -> Position: (3,0)
   -> Number of neighbors: 1
   -> Cost: 2.0
   -> Visited: true
```