Written Assignment Unit 4

Department of Computer Science, University of the People

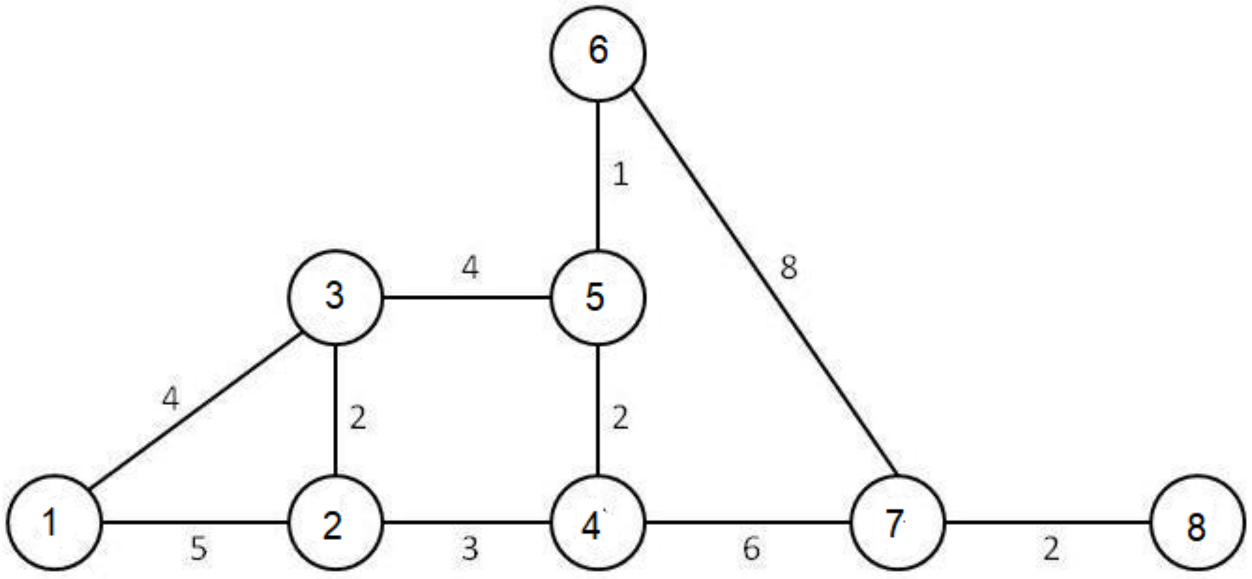
CS 3304-01: Analysis of Algorithms

Instructor Romana Riyaz

July 17, 2024

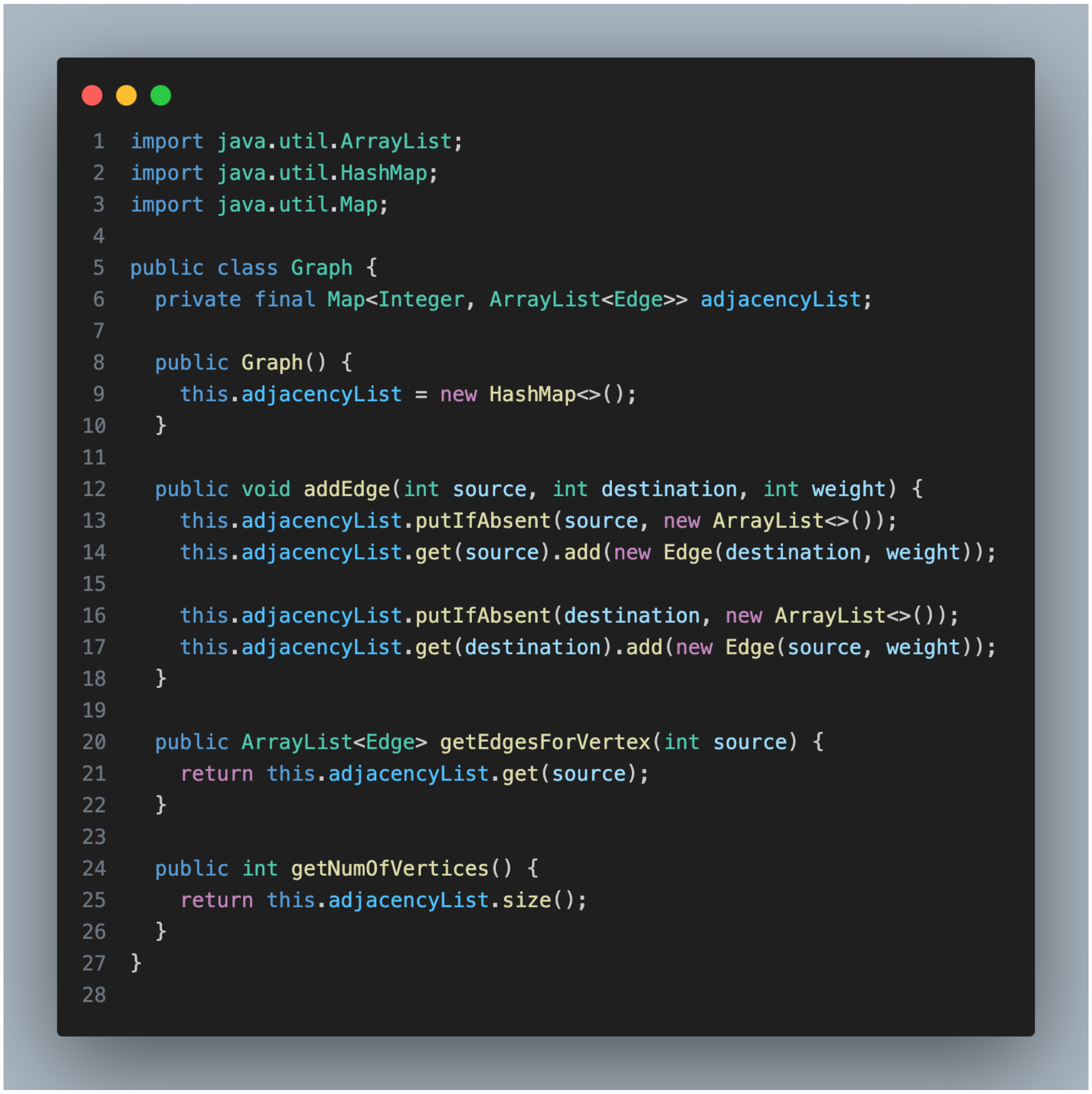
Written Assignment Unit 4

Develop an implementation of Prim's algorithms that determines the MST (Minimum Spanning Tree) of the graph from the Unit 2 assignment that we developed the data structure for.



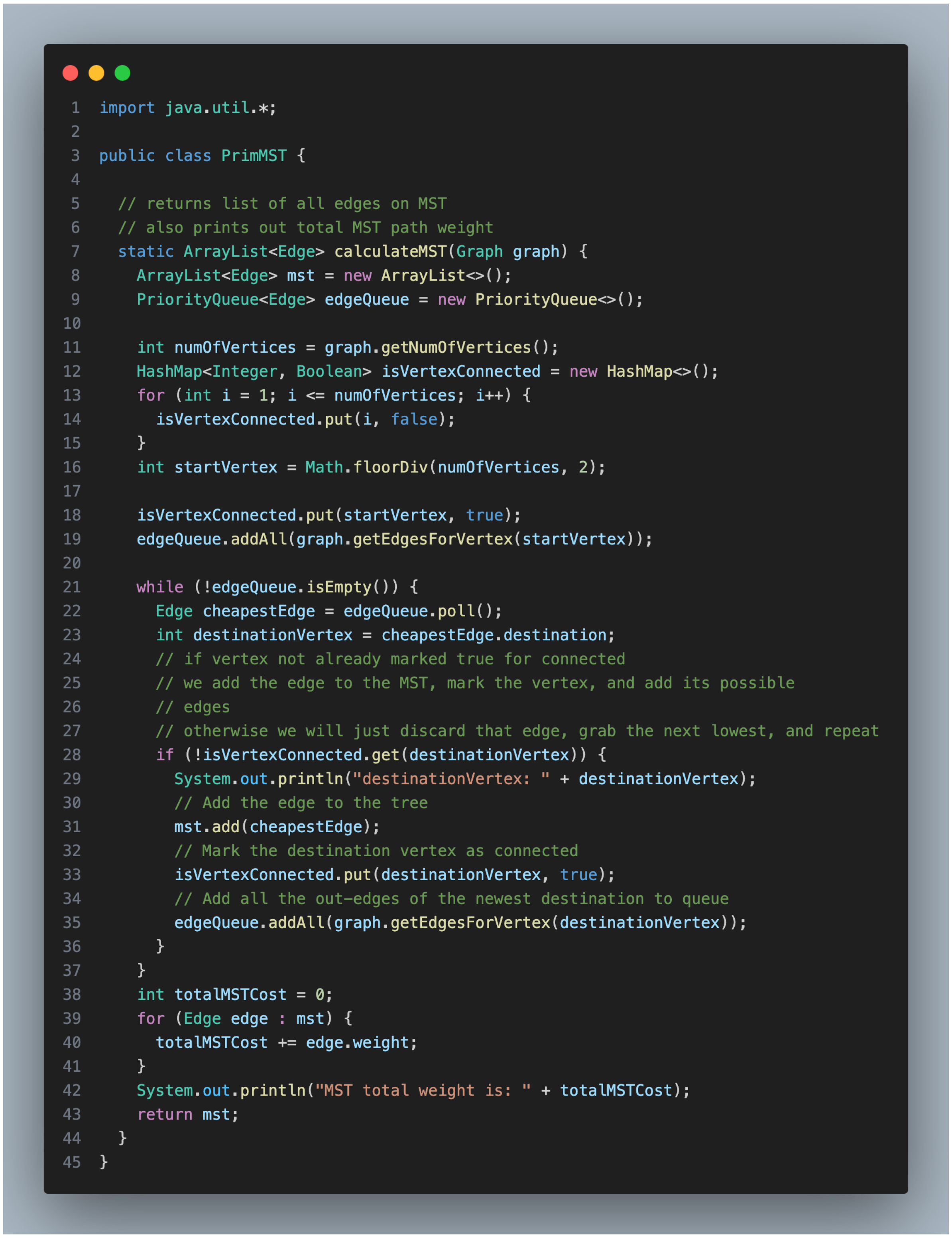
For this assignment, develop an implementation using Java in the Cloud9 environment (or your own Java IDE) that first implements the graph in a data structure and then provides the algorithm that can determine the Minimum spanning tree within this graph in terms of cost. The cost will be the sum of the lengths of the edges that must be traversed. The cost of each edge is represented by the number on the edge. For example, the cost of edge 1,3 is 4 and the cost of edge 6,7 is 8. Your algorithm must output the total cost of spanning the tree as determined by your implementation of Prim's algorithm. The algorithm must produce output which is the total cost of the path.

The first task was creating a custom graph structure in the Java language. Here is the code for my graph structure:



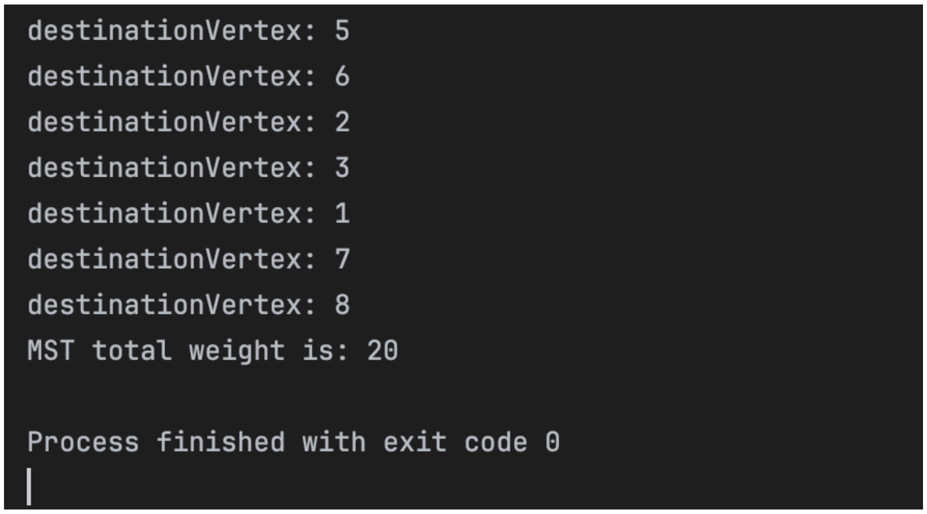
```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;

public class Graph {
  private final Map<Integer, ArrayList<Edge>> adjacencyList;

  public Graph() {
    this.adjacencyList = new HashMap<>();
  }

  public void addEdge(int source, int destination, int weight) {
    this.adjacencyList.putIfAbsent(source, new ArrayList<>());
    this.adjacencyList.get(source).add(new Edge(destination, weight));

    this.adjacencyList.putIfAbsent(destination, new ArrayList<>());
    this.adjacencyList.get(destination).add(new Edge(source, weight));
  }

  public ArrayList<Edge> getEdgesForVertex(int source) {
    return this.adjacencyList.get(source);
  }

  public int getNumOfVertices() {
    return this.adjacencyList.size();
  }
}

```

This graph implementation uses a map called adjacencyList to keep track of every vertex and its adjacent vertices. The key is an integer for the vertex label and the value is a list of all adjacent vertices for that vertex. Every time an edge is added, it updates both the source and destination vertices accordingly using the HashMap method putIfAbsent(). This way, as is necessary, both vertices have reference to one another. The other helper methods are clear cut: getEdgesForVertex() returns a list of all adjacent vertices for a specific vertex and getNumOfVertices() gets the total number of vertices in the graph. Next is the algorithm itself:



```java
import java.util.*;

public class PrimMST {

  // returns list of all edges on MST
  // also prints out total MST path weight
  static ArrayList<Edge> calculateMST(Graph graph) {
    ArrayList<Edge> mst = new ArrayList<>();
    PriorityQueue<Edge> edgeQueue = new PriorityQueue<>();

    int numOfVertices = graph.getNumOfVertices();
    HashMap<Integer, Boolean> isVertexConnected = new HashMap<>();
    for (int i = 1; i <= numOfVertices; i++) {
      isVertexConnected.put(i, false);
    }
    int startVertex = Math.floorDiv(numOfVertices, 2);

    isVertexConnected.put(startVertex, true);
    edgeQueue.addAll(graph.getEdgesForVertex(startVertex));

    while (!edgeQueue.isEmpty()) {
      Edge cheapestEdge = edgeQueue.poll();
      int destinationVertex = cheapestEdge.destination;
      // if vertex not already marked true for connected
      // we add the edge to the MST, mark the vertex, and add its possible
      // edges
      // otherwise we will just discard that edge, grab the next lowest, and repeat
      if (!isVertexConnected.get(destinationVertex)) {
        System.out.println("destinationVertex: " + destinationVertex);
        // Add the edge to the tree
        mst.add(cheapestEdge);
        // Mark the destination vertex as connected
        isVertexConnected.put(destinationVertex, true);
        // Add all the out-edges of the newest destination to queue
        edgeQueue.addAll(graph.getEdgesForVertex(destinationVertex));
      }
    }
    int totalMSTCost = 0;
    for (Edge edge : mst) {
      totalMSTCost += edge.weight;
    }
    System.out.println("MST total weight is: " + totalMSTCost);
    return mst;
  }
}
```

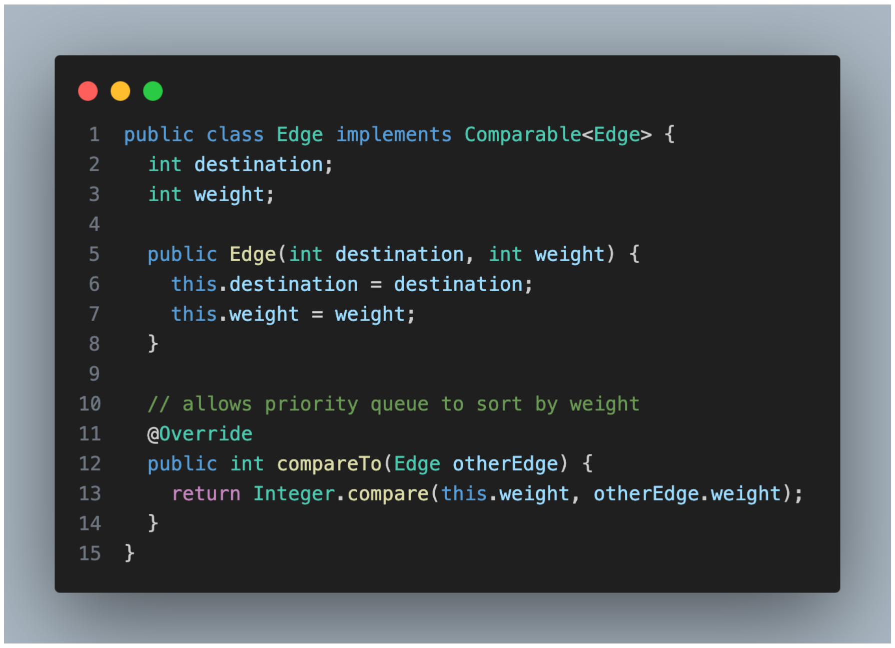
My implementation uses a priority queue and another map. The priority queue is of course automatically sorted by edge weight ascending due to the nature of the class and the edge class's custom comparator (seen below). The HashMap *isVertexConnected* keeps track of whether any one vertex is connected to our running MST. We start all as false of course. We grab a starting vertex by splitting the total number of vertices in half ; we mark that one as connected and add all of its adjacent neighbors to the queue. We keep polling that queue until we find an unconnected vertex. We mark it as connected, add all of its neighbors to the queue, and the process begins again. This continues until the queue is empty of course; at that point our MST is complete. My comments, which aided in building the algorithm, have been left for reference. Here is the output after the program is ran:



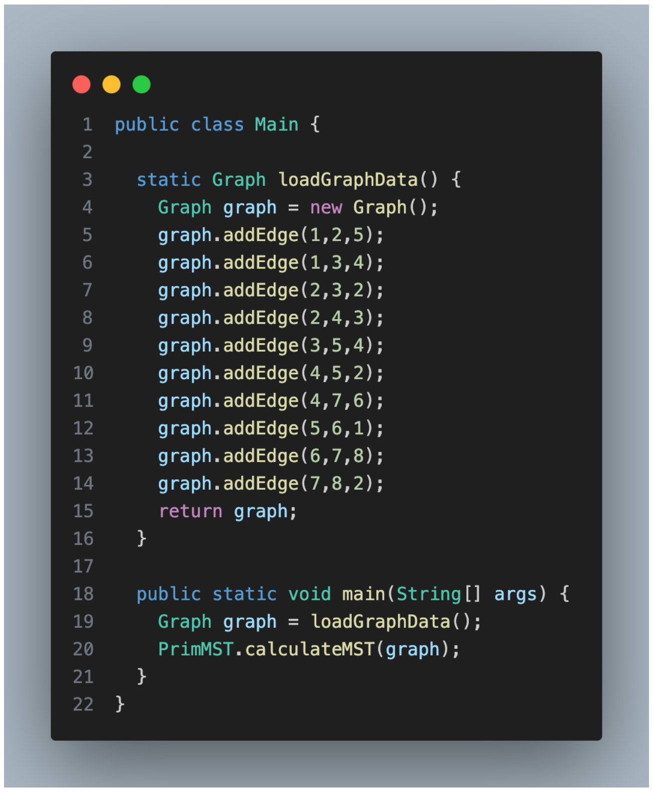
```
destinationVertex: 5
destinationVertex: 6
destinationVertex: 2
destinationVertex: 3
destinationVertex: 1
destinationVertex: 7
destinationVertex: 8
MST total weight is: 20

Process finished with exit code 0
```

As seen in my while loop, my algorithm prints to console every time a new vertex is connected to our MST. This allows the user to see the algorithm following the path as expected by how the algorithm functions. As expected, the MST total weight is 20 units.

For further reference here are my 'Edge' and 'Main' classes as well:



```java
public class Edge implements Comparable<Edge> {
  int destination;
  int weight;

  public Edge(int destination, int weight) {
    this.destination = destination;
    this.weight = weight;
  }

  // allows priority queue to sort by weight
  @Override
  public int compareTo(Edge otherEdge) {
    return Integer.compare(this.weight, otherEdge.weight);
  }
}
```



```java
public class Main {

  static Graph loadGraphData() {
    Graph graph = new Graph();
    graph.addEdge(1,2,5);
    graph.addEdge(1,3,4);
    graph.addEdge(2,3,2);
    graph.addEdge(2,4,3);
    graph.addEdge(3,5,4);
    graph.addEdge(4,5,2);
    graph.addEdge(4,7,6);
    graph.addEdge(5,6,1);
    graph.addEdge(6,7,8);
    graph.addEdge(7,8,2);
    return graph;
  }

  public static void main(String[] args) {
    Graph graph = loadGraphData();
    PrimMST.calculateMST(graph);
  }
}
```

The main class of course simply loads the graph structure with all necessary edges and calls our algorithm method. Edge creates a simple edge object with a destination vertex (which is really just an integer; my graph does not use a vertex object) and a weight. The compareTo override allows the priority queue to properly compare edges by weight when they are entered into the structure. According to Geeksforgeeks (2024), the Big O analysis, when using an adjacency list, is O((V+E)•log•V). This is a very efficient algorithm requiring no double loops or recursion.

References

GeeksforGeeks. (2024, February 9). *Time and Space Complexity Analysis of Prim's Algorithm*. https://www.geeksforgeeks.org/time-and-space-complexity-analysis-of-prims-algorithm/