IFT2015 ASSIGNMENT 3

July 17, 2023

Massimo Pietracupa (20206087) and Vincent Hoang (20183549)

1: Time complexity analysis in Big O notation:

We'll consider the number of vertices in the undirected graph as *n* when considering time complexity

We will ignore the processing of the file, as this process is will simply loop through every line in the file and store the information into corresponding object arrayLists, which is an operation of O(1), indicating that the time complexity of this section is directly related to the number of lines of the file, O(k), where k is the number of lines. We can also replace k with V + E, where V is the number of vertices and E is the number of Edges.

The second section will be the processing of the information. We first create a new Carte object that consumes a vertices ListArray. The Carte Object contains a Graph object, which consumes the vertices ListArray and stores these into a HashMap with the vertice name as the key and an empty *Edge* ArrayList, which is a structure that contains edge information. Vertices are first sorted to abide by the output restrictions of the assignment. Hence, without a sort we can expect a time complexity of O(V), though with the sort we can expect a time of **O(VlogV)**, as the collections.sort utilizes a form of merge sort, where V is the number of vertices.

```
Carte carte = new Carte(vertices);
for (Edge edge : edges) carte.addRue(edge);

List<Edge> mst = carte.primJarnikAlgorithm();
//Collections.sort(mst);
Collections.sort(mst, Edge.vertexComparator);

for (String vertex : vertices)
{
    bufferedWriter.write(vertex + "\n");
}

// Print the minimum spanning tree
int totalWeight = 0;
for (Edge edge : mst)
{
    bufferedWriter.write(edge.getName() + "\t" + edge.getVertex0() + "\t" +
    totalWeight += edge.getWeight();
}
bufferedWriter.write("---\n");
bufferedWriter.write(totalWeight + "\n");
```

```java
public class Carte
{
    Graph graph;
    HashMap<String, Edge> originalEdges = new HashMap<>();

    public Carte(ArrayList<String> vertices)
    {
        //System.out.println(vertices.size());
        this.graph = new Graph(vertices);
    }
```

```java
public class Graph
{
    ArrayList<String> vertices;
    HashMap<String, ArrayList<Edge>> adjacencyList;

    public Graph(ArrayList<String> vertices)
    {
        this.vertices = vertices;
        Collections.sort(this.vertices);
        adjacencyList = new HashMap<String, ArrayList<Edge>>();

        for (String vertex : this.vertices)
        {
            adjacencyList.put(vertex, new ArrayList<Edge>());
        }
    }
```

Edges are then added to the empty adjacencyLists that were initialized before. Getting and putting inside a hashmap has time complexity of O(1). This means that the time complexity is directly proportional to the number of edges between the nodes. If we represent that by *E*, then the time complexity will be O(E).

```java
public void addEdge(String name, String vertex0, String vertex1, Integer weight)
{
    ArrayList<Edge> e0 = adjacencyList.get(vertex0);
    e0.add(new Edge(name, vertex0, vertex1, weight));    // Add edge
    adjacencyList.get(vertex1).add(new Edge(name, vertex1, vertex0, weight));   // Add edge other direction
}
```

We now perform the Prim Jarnik Algorithm. The initialization part involves creating data structures, such as the visitedNodes set, priorityQ priority queue, and minimumSpanningTree list. The time complexity for this part is O(1) since it does not depend on the size of the graph.

We then proceed to building the priority queue. The algorithm starts by adding the edges that are touching the starting vertex to the priority queue. The time complexity of this step depends on the

number of edges touching the starting vertex (E) and the edges already in the priority queue (n), which can be denoted as O(Elogn).

The while loop iterates until the priority queue has been completely emptied. In each iteration, the algorithm removes the minimum-weight edge from the priority queue, (which takes into account the weight of the Edge and the letter priority of the vertex), then adds it to the minimum spanning tree, and finally marks the new vertex as visited. The new edges that are next to the visited vertices are added to the priority queue. The number of iterations in the while loop depends on the number of edges in the minimum spanning tree, which can be denoted as O(E). Once again however, we are adding the edges to the priority queue at the end of the loop, with a time complexity of O(Elogn), resulting in a total time complexity of O(2Elogn), which can be simplified to O(Elogn).

Combining the above steps, the overall time complexity of the function is O(Elogn), however n can be at worst case E, indicating that the final time complexity is **O(ElogE)**, where E is the number of edges in the graph.

```java
public List<Edge> primJarnikAlgorithm()
{

    Set<String> visitedNodes = new HashSet<>(); // O(1)
    PriorityQueue<Edge> priorityQ = new PriorityQueue<>((edge1, edge2) -> { // O(1)
        if (edge1.getWeight() != edge2.getWeight()) {
            return Integer.compare(edge1.getWeight(), edge2.getWeight());
        } else {
            return edge1.getVertex0().compareTo(edge2.getVertex0());
        }
    });

    List<Edge> minimumSpanningTree = new ArrayList<>(); // O(1)

    String startVertex = graph.getVertices().get(0); // O(1)
    visitedNodes.add(startVertex); // O(1)

    List<Edge> edges = graph.getAdjacencyList().get(startVertex); // O(1)
    priorityQ.addAll(edges); // O(Elogn) where E is the number of edges and n is the number of entries in queue


    while (!priorityQ.isEmpty()) // Worst case O(E)
    {
        Edge minEdge = priorityQ.remove(); // O(1)
        String v0 = minEdge.getVertex0(); // O(1)
        String v1 = minEdge.getVertex1(); // O(1)

        // Skip if both vertices are already visitedNodes
        if (visitedNodes.contains(v0) && visitedNodes.contains(v1)) // O(1)
        {
            continue;
        }

        // Add the edge to the minimum spanning tree
        minimumSpanningTree.add(minEdge); // O(1)

        // Mark the new vertex as visited
        String newVertex = visitedNodes.contains(v0) ? v1 : v0; // O(1)
        visitedNodes.add(newVertex); // O(1)

        // Add all edges incident to the new vertex to the priority queue
        edges = graph.getAdjacencyList().get(newVertex); // O(1) because of hashmap
        priorityQ.addAll(edges); // O(Elogn) where k is the number of edges and n is the number of entries in queue
    }

    return minimumSpanningTree;
}
```

Combining everything together we have the following:

O(V+E) + O(VlogV) + O(ElogE) **=** O(V + E + VlogV + ElogE) where:

- V is the number of vertices
- E is the number of edges

Though we can simplify this further by keeping the larger terms:

O(V + E + VlogV + ElogE) = O(VlogV + ElogE)

Finally, since V is typically smaller than E, we can ignore V and just keep the higher order term:

O(VlogV + ElogE) = **O(ElogE)**