Algorithms and Time Complexity Project 2

Shortest Routes

Indrit Caca, Ronaldo Gjini

June 25, 2021

Contents

1 Introduction

This report paper represents the solution and the implementation of a program that takes a graph inputted from the user as a text file. Such file text is then read and processed from the program and transformed into useful information about the graph such as edges. We consider the problem that is trying to be solved in this assignment to be finding whether the graph inputted from the user is a DAG (Directed Acyclic Graph) or not. Furthermore, regardless of whether the graph is a DAG or not it is important in this assignment to find out the shortest path and shortest path cost for these cases. As for the requirement of this assignment, we have to implement three algorithms for checking out whether the graph is DAG or not and also for finding the shortest path and the shortest path cost for the cases where the graph is a DAG and where the graph is not a DAG.

2 Solution

To implement the solution for the Shortest Routes assignment we have created 5 clasees: **Main**, **Vertex**, **Edge**, **Graph** and **GraphLibrary** . Furthermore, in our solution we have created folder called **graphFiles** where the graph representations in text files are saved and retrieved for usage in the program.

2.1 Main Class

The Main class is the running class of our implemented solution. In our Main class it is entered a string path which represents the relative path where the text file is saved. This string is then inputted as a parameter for a static method from the GraphLibrary which will read the contents of the file from the specified file, generate a graph with the number of edges and vertices and then add the newly created edges read from the file into the graph. Finally, after the graph is created we supply it and the source vertex (which in the requirements of the assignment is the vertex 0). The result will the output that specifies whether the inputted graph is a DAG or not and the shortest path and the shortest path cost from the source vertex to all the other vertices of the graph.

2.2 Edge Class

The Edge class is a simple class that is needed when we want to add edges to the graph. It contains 2 integer values which represent the source and destination vertex values and a double value for containing the weight of the edges and also a constructor for creating an instance of the class as well as the getters and setters for the values mentioned above.

2.3 Vertex Class

The Vertex class is also a simple class that is needed when we add the vertices into a priority queue. It contains 1 integer value which represent the numbered id of the vertex and a double value for containing the weight of the vertex and also a constructor for creating an instance of the class as well as the getters and setters for the values mentioned above.

2.4 Graph Class

The Graph class is a simple class that is needed at the main class of the program. The instances of this class are constructed from the input of the text files and used in the algorithms implemented in the GraphLibrary. In our solution we have implemented the representation of the edges (which means all vertices and the neighbor vertices) in adjacency list form. The reason why we implemented the adjacency list instead of the adjacency matrix is the fact that the adjancency list is better than adjacency matrix when comparing the time and space complexity. Given the number of vertices $n = |V|$, and the number of edges $e = |E|$, where $E \subseteq N^2$, for the adjacency matrix we would need to create a $n * n$ matrix which would hold true/false values depending on whether an edge exists. The same scenario is presented also when we want to build the matrix because we would need to enter as much as n^2 values regardless if an edge exists or not. As such both the time and space complexity of the adjacency matrix are $O(n^2)$ or $O(|V|^2)$. In case of the adjacency list however, a different and much more efficient scenario is presented. Using the information that we explained above, we can create adjacency lists using an ArrayList of vertices, each having an ArrayList of neighboring vertices (which is represented by the edges between these two vertices). As such the time complexity of building adjacency lists is $O(e)$ or $O(|E|)$ and the space complexity of building adjacency lists is $O(v + e)$ or $O(|V| + |E|)$ which is much better than the adjacency matrix. Even if we are to compare the worst case which would be a dense graph, the time and space complexities for both adjacency matrix and adjacency list would be $O(n^2)$. So adjacency lists offer us a better option in the case of sparse graphs and an equal option in the case of dense graphs regarding time and space complexities. After evaluating the data structure we move on to the properties of this class. This class contains 2 integer values for storing the number of edges and the number of vertices as well an adjacency list represented as an ArrayList of ArrayLists of type Edge, a constructor for initializing an instance of the class and the getter and setter methods.

2.5 GraphLibrary Class

The GraphLibrary class is the class where the algorithms required from the assignment requirements specification paper are created. It contains methods for checking if the input graph is a DAG or not, it contains a depth first search algorithm implementation and shortest path and shortest path cost algorithms for the general and the specific case of graphs. A more detailed time complexity and desciption analysis is perfomed on the next section.

3 Algorithms Implementation and Time Complexity Discussion

3.1 Explore Method

3.1.1 Explore Java Implementation

```
public static int explore(Graph graph, int source, int clock,
    int[] previsitClock, int[] postVisitClock,
    ArrayList<Integer> vertexReverseOrder, boolean[] visited){

     visited[source] = true;
     previsitClock[source] = clock++;
     for (Edge destination : graph.adjancencyLists.get(source)){
         if(!visited[destination.getDestinationVertex()]){
             clock = explore(graph,
                 destination.getDestinationVertex(), clock,
                 previsitClock, postVisitClock,
                 vertexReverseOrder, visited);
         }
     }
     vertexReverseOrder.add(source);
     postVisitClock[source] = clock++;
     return clock;
 }
```

3.1.2 Explore Time Complexity Analysis

Explore method that we have implemented is an implementation of the Depth First Search algorithm as it was explained in the textbook of the course. The data structure used for this implementation is an arraylist. As such the time complexity of accessing an arraylist element with a given index is $O(1)$ and the time complexity of deleting an arraylist element using its index is at worst case $O(n)$ as all the other values would be shifted. As such, it is a efficient to create an adjacency list representation of the graph using the arraylist. Firstly, the method takes 7 parameters: the graph, the source vertex of the graph, the clock which is a way of representing the steps the algorithm takes to visit each connected vertex in the graph and which will be returned as the output of the method, the previsit and the postvisit arrays that hold the clock at which a vertex was visited from the algorithm on the algorithm's way in and its way out, vertexReverseOrder which keeps an array of vertices starting from the vertex with the least postVisit clock and ending with the source vertex and a visited array which keeps track whether the vertices have been visited or not. Starting the time complexity analysis, the first two operations which set the source index in both the visited and previsit arrays with true value and an incrementation respectively have a time complexity of $O(1)$. Next we have a for loop which is going to iterate over all the neighbours of the given source vertex. At the worst case, there could be up to $e = |E|$ edges from the source vertex so the number of times this loop will iterate would be e. We perform a check inside the for loop, to see if the destination vertex is visited and if not a recursive call would be made. After the loop terminates, we perform an arraylist insertion, an increment function and we also return a value. All of these have a time complexity of $O(1)$. Starting from the vertex with the least postVisit which will be the base case that will end recursion we can see that the time complexity is $1+1+1+1+1 = O(1)$ for a single recursive call. Since we said earlier that the loop with iterate $e - times$, that means that the number of recursive calls would be e so ultimately a depth first search from a given source vertex to all its neighbors would have a time complexity of $O(e)$. So the time complexity of the method implemented above is $O(e)$. This would hold true for average case which would be sparse graphs and the worst case which would be dense graphs.

3.2 CheckDirectedAcyclicGraph Method

3.2.1 CheckDirectedAcyclicGraph Java Implementation

```
public static boolean checkDirectedAcyclicGraph(Graph graph){

    int numberOfVertices = graph.getNumVertices();
    int clock = 0;
```

```
        boolean[] visited = new boolean[numberOfVertices];
        int[] preVisitClock = new int[numberOfVertices];
        int[] postVisitClock = new int[numberOfVertices];

        for (int vertexID = 0; vertexID < numberOfVertices;
            vertexID++){
            if(!visited[vertexID]){
                clock = DepthFirstSearch(graph, vertexID, clock,
                    preVisitClock, postVisitClock,
                    vertexReverseOrder, visited);
            }
        }

        for (int vertexId = 0; vertexId < numberOfVertices;
            vertexId++){
            for (Edge edge : graph.adjancencyLists.get(vertexId)) {
                if (postVisitClock[vertexId] <=
                    postVisitClock[edge.getDestinationVertex()]){
                    return false;
                }
            }
        }
        return true;
    }
```

3.2.2 CheckDirectedAcyclicGraph Time Complexity Analysis

CheckDirectedAcyclicGraph method is the method which takes a graph as the input and checks if the inputted graph is a DAG or not. To do that, it checks if the graph contains any back edges as the indication of back edges in a graph means that the graph is not a DAG. A back edge is defined like:

$$u \to v \Rightarrow postVisit[u] \leq postVisit[v]$$

To do that we need to do an entire depth first search traversal of the graph in order to record the postVisit markers of each vertex. Then we can compare these postVisit markers to check if any of the vertices is actually forming a back edge. As with the previous method, the data structure of choice is the arraylist due to its efficiency in time complexity for operations like the insertion and retrieval of the elements. Firstly we initialize the number of vertices by taking the value from the inputted graph. Next we initialize the clock at 0, and this clock will be incremented throughout the depth first search traversal. Next in line is the initialization of the arrays visited, preVisitClock and postVisitClock which have all the size of the vertices that the graph contains. The first array holds the true/false values of whether a vertex has been visited and the next two arrays hold the preVisit and postVisit time. The time complexity of all these operations is $O(1)$. Next we have to perform the depth first traversal on the entire graph. This is done by iterating over each vertex, which we use as the source vertex for calling the depth first search method which we discussed earlier. Since the loop will iterate $n = |V|$ times, as that is the number of vertices present in the graph, it will need to call the depth first seach method n times. We discussed earlier that the time complexity of the a depth first search method is $O(e)$, so the time complexity of the entire graph depth first search traversal is going to be $O(v + e)$ or $O(|V + |E|)$. Furthermore, we also need to perform a check on every neighbor of each vertex to see if there is any existing back edge. In a way, the depth first search traversal and the back edge checking for loop resemble a lot. Both have an outer for loop that iterates n times as that is the number of vertices, and the inner for loop iterates e times as that is the number of neighbors. However, when we are checking for back edges, we need to perform a comparison operation and return a boolean value, both of which have a time complexity of $O(1)$. As a result the time complexity of the for loop that checks for the back edges is $O(v + e)$ after the lower-order terms are dropped, just like the depth first search traversal above. Combining all the time complexities we get that the time complexity is: $5 + v + e + v + e = 2v + 2e + 5 = O(v + e)$ or $O(|V + |E|)$ when the lower-order terms are dropped for the case when the graphs are sparse. But, in the worst case, when the graphs are dense, $|E|$ can be as big as $|V|^2$ so the time complexity for dense graphs with this algorithm would be $O(|V|^2)$.

3.3 DirectedAcyclicGraph Method

3.3.1 DirectedAcyclicGraph Java Implementation

```
public static void DirectedAcyclicGraph(Graph graph, int
    sourceVertex){
        if (GraphLibrary.checkDirectedAcyclicGraph(graph)) {
```

```java
            System.out.println("Inputted Graph is a Directed Acyclic
                Graph");
            calculateDagShortestPath(graph, sourceVertex);
        }
        else {
            System.out.println("Inputted Graph is not a Directed
                Acyclic Graph");
            calculateNonDagShortestPath(graph, sourceVertex);
        }
    }
```

3.3.2 DirectedAcyclicGraph Time Complexity Analysis

This method will take the graph and the source vertex as input. It only performs a check if the graph is a DAG or not which we said has a worst case time complexity of $O(|V|^2)$ when the graph is dense and the average case time complexity of $O(|V + |E|)$ when graph is sparse. If the inputted graph is a DAG, the time complexity if it is sparse it is going to be $|V| + |E| + |V| + |E| = 2(|V| + |E|) = O(|V + |E|)$ and if it is dense it is going to be $|V|^2 + |V|^2 = 2(|V|^2) = O(|V|^2)$. There is also the case where the graph is a non-DAG. As we mentioned, calculating non-DAG shortest path has a average time complexity of $O((|V| + |E|)log|V|)$ and a worst time complexity of $O(|V^2|(log|V|)$. In the same way, if the inputted graph is a non-DAG, the time complexity if it is sparse it is going to be $|V| + |E| + (|V| + |E|)log|V| = (log|V| + 1)(|V| + |E|) = O((|V| + |E|)log|V|)$, and in the case it is a dense graph it is going to be $|V|^2 + |V^2|log|V| = (log|V| + 1)(|V|^2) = O(|V^2|log|V|)$

3.4 DagLinearization Method

3.4.1 DagLinearization Java Implementation

```java
public static ArrayList<Integer> dagLinearization(){
    ArrayList<Integer> topologicalOrder = new ArrayList<>();
    for (int j = vertexReverseOrder.size()-1; j>=0; j--)
    {
        topologicalOrder.add(vertexReverseOrder.get(j));
    }
```

```
        return topologicalOrder;
    }
```

3.4.2 DagLinearization Time Complexity Analysis

This method is used for implementing the linearization of a given DAG. This linearization is based on the vertexReverseOrder array specified at the beginning of the class. This array it is populated in the depth first search method implemented above. It inserts into the array, starting from the first index until the end, all the vertices based on the postVisit clock in a decreasing order. The dagLinearization method simply reverses the order of this array, which leaves us with an array list where the first node is the source node, and the last node is a sink, which is the linearization implementation that we want. The method creates an arraylist which has time complexity of $O(1)$. Furthermore, there is a for loop which traverses the entirety of the vertexReverseOrder array of size $|V|$, and adds the vertexes to the new arraylist, with a time complexity of $O(1)$. Thus, the entire cost of the loop is $|V| * 1 = O(|V|)$, and ultimately, after dropping the lower order terms, we have a time complexity of $O(|V|)$ for sparse graphs. The same thing can be said also about dense graphs, as the addition of new vertices would linearly increase time complexity so we have a time complexity of $O(|V|)$ also for dense graphs.

3.5 CalculateDagShortestPath Method

3.5.1 CalculateDagShortestPath Java Implementation

```
  public static void calculateDagShortestPath(Graph graph, int
      source){
       double[] distance = new double[graph.getNumVertices()];
       ArrayList<Integer> graphLinearization = dagLinearization();
       int[] previous = new int[graph.getNumVertices()];

       for(int i=0; i<distance.length; i++)
       {
           distance[i] = infinity;
       }
       distance[source] = 0;
```

```
        previous[source] = -1;

        for (int i=0; i<graphLinearization.size(); i++){
            int nextVertex = graphLinearization.get(i);
            if(distance[nextVertex] != infinity){
                for (Edge neighbor :
                    graph.adjancencyLists.get(nextVertex)) {
                    int destination =
                        neighbor.getDestinationVertex();
                    double weight = neighbor.getWeight();
                    if(distance[destination] >= distance[nextVertex]
                        + weight){
                        distance[destination] = distance[nextVertex]
                            + weight;
                        previous[destination] = nextVertex;
                    }
                }
            }
        }
        showShortestPathCost(graph, source, distance, previous);
    }
```

3.5.2 CalculateDagShortestPath Time Complexity Analysis

.

CalculateDagShortestPath is a method that finds the shortest path for a given Directed Acyclic Graph (DAG). It takes as parameters the graph itself, as well as the starting vertex (source). The method then creates an array which keeps track of the distance of each vertex from the source, which takes constant time $O(1)$. Then we represent the graph in an ArrayList in an linearized fashion, considering that this graph has no cycles (DAG). The dagLinearization method which is used for the linearization just as it was mentioned on its analysis, takes $O(|V|)$. Then we create an array which keeps track of the previous vertices for each node, and with previous we specify the previous vertex in the path from the source vertex to a specified vertex, with time complexity of $O(1)$. Then, we set the initial distance for each vertex to ∞, by iterating the array through a for loop. The loop is iterated v times, where $v = |V|$, and each iteration assigns the ∞ to each element of the array, which takes $O(1)$, so ultimately the entire cost of the outer loop without taking in consideration inner parts is $|V| * 1 = O(|V|)$. Then the distance of the source node is set to 0 and the previous vertex is set to -1. These commands take $O(1)$ both. The we have a for loop which iterates over the linearized graph representation, which has a number of $|V|$ vertices. We then retrieve the current vertex on the ArrayList and check if it is not equal to infinity, both in $O(1)$. If it does not, we read the next value present in the adjacency list and go through all the neighbor edges of this particular vertex by using a for loop, which takes $O(|E|)$. We then retrieve the destination of the current edge, and the weight of this edge, in $O(1)$. Then we have an if condition that checks whether the weight of the destination node is higher than the distance of the current node + the weight of the edge. If it is not, we update the distance of the node with the new values, and set the new node on the array that keeps track of the previous values. All these operations inside the for loop take constant time $O(1)$, so the inner loop takes $|E| + 5 = O(|E|)$. This is the end of the entire for loop, which ultimately has a time complexity of $|V| + |E| = O(|V| + |E|))$ when considering both inner and outer loops. Then we call the **showShortestPathCost** method, which simply prints the contents of the arrays in $O(|V|)$. Ultimately, the time complexity of the method after dropping the lower order terms is $|V| + |V| + |E| = O(|V| + |E|)$. This is however the average case, when the graphs are sparse. But, in the worst case, when the graphs are dense, $|E|$ can be as big as $|V|^2$ so the time complexity for dense graphs would be $O(|V|^2)$.

3.6 CalculateNonDagShortestPath Method

3.6.1 CalculateNonDagShortestPath Java Implementation

```
public static void calculateNonDagShortestPath(Graph graph, int
    source){
     PriorityQueue<Vertex> minHeap = new
         PriorityQueue<>(Comparator.comparing(vertex ->
         vertex.getWeight()));
     double[] distance = new double[graph.getNumVertices()];
     boolean[] completed = new boolean[graph.getNumVertices()];
     int[] previous = new int[graph.getNumVertices()];
     for(int i=0; i<distance.length; i++)
     {
         distance[i] = infinity;
     }
     distance[source] = 0;
     minHeap.add(new Vertex(source, 0.0));
     previous[source] = -1;
     completed[source] = true;

     for (int i=1; i<graph.getNumVertices()-1; i++) {
         minHeap.add(new Vertex(i, infinity));
     }
     while (!minHeap.isEmpty()){
         int vertexId = minHeap.poll().getId();
         for (Edge neighbor :
             graph.adjancencyLists.get(vertexId)) {
             int destination = neighbor.getDestinationVertex();
             double weight = neighbor.getWeight();
             if(!completed[destination]){
                 if(distance[destination] >= distance[vertexId] +
                     weight) {
                     distance[destination] = distance[vertexId] +
                         weight;
                     previous[destination] = vertexId;
                     minHeap.add(new Vertex(destination,
                         distance[destination]));
                 }
             }
         }
         completed[vertexId] = true;
     }
     showShortestPathCost(graph, source, distance, previous);
 }
```

3.6.2 CalculateNonDagShortestPath Time Complexity Analysis

CalculateNonDagShortestPath is a method that finds the shortest path for a given non-DAG. It takes as parameters the graph itself, as well as the starting vertex (source). The method then creates a PriorityQueue with a comparator which takes the vertex weight as an input and places the vertex on the PriorityQueue based on that input. Since we are implementing a MinHeap, the vertex with the smallest weight will be placed as the root in the heap. Then we create an array which keeps track of the distance of each vertex from the source, an array which stores the completed vertices (completed in the sense that the vertex's final distance from the source has been computed), and an array that stores the previous node in the path of the vertex from the source vertex. All these operations have a time complexity of $O(1)$. Then, we set the initial distance for each vertex to ∞, by iterating the array through a for loop. The loop is iterated v times, where $v = |V|$, and each iteration assigns the ∞ value to each element of the array, which takes $O(1)$, so ultimately the entire cost of the loop is $|V| * 1 = O(|V|)$. Then the distance of the source node is set to 0, based on the requirements of the assignment. We also perform an addition to the MinHeap which in our case has a time complexity of $O(1)$ as the heap does not contain any other element at the moment. Since the source node does not have any other previous nodes in its path, we set the previous vertex to -1. We also set the source index in the computed array as true. All these operations take $O(1)$. Next we populate the MinHeap with ∞ values for each vertex. Adding these vertices to the MinHeap is the best case scenario since all the vertices have the same ∞ which means that do not have to swap with their parent nodes, so the total time complexity is $1 * |V| = O(|V|)$. However, in the worst case, adding these vertices would have a time complexity of $O(|V|log|V|)$. Then we have a while loop, which checks if the PriorityQueue is empty or not. From here we will remove the minimum vertex of the heap and take its vertexID. This has a time complexity of $O(log|V|)$. Since there are $|V|$ vertices, we would need to perform $|V|$ minimum deletions with a total time complexity of $O(|V|log|V|)$. Next we will check each neighboring edge from from the vertex ID we extracted. From these neighboring edges we perform the same actions we performed earlied in the DAG which are getting the destination and the weight of the edge which have a time complexity of $O(1)$. Then we have an if condition that checks whether the weight of the destination node is higher than the distance of the current node + the weight of the edge. If it is not, we update the distance of the node with the new values, and set the new node on the array that keeps track of the previous values with a time complexity of $O(1)$ and finally we perform an insertion which has a time complexity of $O(log|V|)$. So the total cost for the insertions in our loop is $(|V| + |E|) * log(|V|) = O((|V| + |E|)log|V|)$. Adding the MinHeap deletions and the insertions together we get a time complexity of $|V|log|V| + (|V| + |E|) * log(|V|) = O((|V| + |E|)log|V|)$. This is the average and the best case scenario, where the graph is sparse. However, in the cases when the graphs are dense, we mentioned earlier that $|E|$ can be as big as $|V|^2$ so in that case the time complexity for this implementation would be $O(|V^2|log|V|)$.

3.7 ShowShortestPathCost Method

3.7.1 showShortestPathCost Java Implementation

```
    public static void showShortestPathCost(Graph graph, int
        source, double[] distance, int[] previous){
     System.out.println("The distance from source vertex " +
         source + " to other vertices is shown below:");
     for(int vertex = 0; vertex < graph.getNumVertices();
         vertex++){
         System.out.print("Distance from " + source + " to " +
             vertex + " is: ");
         if(distance[vertex] == infinity){
             System.out.println("Infinity\tUnreachable Path");
         }
         else{
             System.out.print(distance[vertex]);
             System.out.print("\t\tPath from " + source + " to "
                 + vertex + " is: ");
             showShortestPathRoute(vertex, previous);
         }
     }
}
```

3.7.2 showShortestPathCost Time Complexity Analysis

The showShortestPathCost method takes as parameters the graph, source, the distance array, the previous path array, and prints the shortest path for each vertex from the source, as well the cost of the path. The method begins with a print operation, which has time complexity of $O(1)$. Next we iterate through all the vertices of the graph, and check if the distance is infinity or not. If the distance is not infinite, we print the cost of the path, as well as the path by calling the showShortestPathRoute method, which in itself has a time complexity of $O(|V|)$. Since the for loop will iterate $|V|$ times, if there is a path from the source to the destination vertex it will have a time complexity of $|V| * |V| = O(|V^2|)$. This would be the case for both sparse and dense graphs.

3.8 ShowShortestPathRoute Method

3.8.1 ShowShortestPathRoute Java Implementation

```
    public static void showShortestPathRoute(int current, int[]
        previous){
    ArrayList<Integer> route = new ArrayList<>();
    while (current >= 0){
        route.add(current);
        current = previous[current];
    }
    for (int element = route.size()-1; element >= 0; element--){
        if(element == 0){
            System.out.print(route.get(element));
        }
        else{
            System.out.print(route.get(element) + " -> ");
        }
    }
    System.out.println();
}
```

3.8.2 ShowShortestPathRoute Time Complexity Analysis

This method finds the entire path from the source vertex to a specific vertex. It takes the current vertex id and the array of previous vertices in the path from the source to the specified vertex as parameters. It perform an array initialization with a time complexity of $O(1)$. Then it will iterate over an array which at worst case will contain $|V|$ elements (for example it might be the case of an array which passes through all the vertices of the graph to reach the source vertex). Inside that while loop we have an arraylist insertion and a value setting so the entire while loop has a time complexity of $O(|V|)$. The following for loop is almost identical in time complexity as it will iterate $|V|$ times, and its operations inside its iterations are a check and a print operations, both with a time complexity of $O(1)$, which again makes the for loop with a time complexity of $O(|V|)$. Adding both this loops together $|V| + |V| = 2|V| = O(|V|)$. Again, this would be the case for both sparse and dense graphs.

3.9 ReadFile Method

3.9.1 ReadFile Java Implementation

```
public static Graph readFile(String path){
int numOfVertices = 0;
int numOfEdges = 0;
ArrayList<Edge> edges = new ArrayList<>();

ArrayList<String> fileLines = new ArrayList<>();
try {
    Scanner input = new Scanner(new File(path));
    while (input.hasNextLine()){
        String inputLine = input.nextLine();
        if(inputLine.startsWith("#")){
            continue;
        }
        fileLines.add(inputLine);
    }
    input.close();
}  catch (FileNotFoundException fe){
   fe.printStackTrace();
}

String[] graphSubstrings = fileLines.get(0).split(" ");
numOfVertices = Integer.parseInt(graphSubstrings[0]);
numOfEdges = Integer.parseInt(graphSubstrings[1]);

for (int i = 1; i < fileLines.size(); i++){
        String[] edgeSubstrings = fileLines.get(i).split("
            ");
        int source = Integer.parseInt(edgeSubstrings[0]);
        int destination =
            Integer.parseInt(edgeSubstrings[1]);
        double weight =
            Double.parseDouble(edgeSubstrings[2]);
        edges.add(new Edge(source, destination, weight));
}
return new Graph(edges, numOfVertices, numOfEdges);
}
```

3.9.2 ReadFile Time Complexity Analysis

The method implemented here performs the read of the text file which contains graph information, which is supplied as path in the parameters. Fristly we perform 2 assignment operations and 2 initializations of arraylists, all with time complexity of $O(1)$. Next we get insert the file path as input with time complexity of $O(1)$. Then we need to read the file lines. Assuming the file lines are represented with a letter n, where $n = filelines$, then the while loop will iterate through all the lines n-times to insert the lines into the arraylist which has a time complexity of $O(1)$ so the time complexity is going to be $n * 1 = O(n)$. Since the input is supposed to be formatted, the time complexity for finding and splitting each lines, for setting the number of vertices and setting the number of edges is also going to be $O(1)$. Again we have a for loop that is going to iterate n-times where n is the number of file lines in the input text. Inside the loop, we perform a substring split, 3 assignment operations and an arraylist insertion. So inside each iteration of the loop we have a time complexity of $(1 + 1 + 1 + 1 + 1) * n = 5 * n = O(n)$. Finally a graph is returned which has a time complexity of $O(1)$. Adding all the time complexities we get $3 + n + n = 2n + 3 = O(n)$ after the lower order terms are dropped, with n as file lines.

4 Summary

To conclude this report, we have shown in this implementation a solution for checking whether an inputted graph is a DAG or a non DAG. Based on the result of this check it is going to be implemented a algorithm that finds the shortest path and the shortest path cost for each type mentioned above. The implemented data structure is the adjacency list, except for the calculation of the non-DAG which also implements a PriorityQueue. In this assignment were used algorithms mentioned in the book like the Depth First Search and

Dijkstra Shortest Path algorithms and they were implemented in an efficient implementation.

5 Notes

Since it was a assignment made by a pair of members it is not possible to separate which work is done from specific member. The project was done from both the members of the group working together. And as such no group member did any specific individual work.