Faculty of Engineering and Architecture

Department of Computer Science

Course: Algorithm and Complexity

Semester: Spring 2023

Coursework: Paths in Graphs

Prepared by: Sofi Çapi & Markela Mykaj

Instructor: Elton Ballhysa, MSc.

Date: 11 June 2023

Table of Contents

I. Problem Statement

The purpose of this assignment is to find the shortest path between vertices in directed cyclic graphs and directed acyclic graphs, known as DAG. This coursework is implemented using Java programming language. The program was implemented using JDK 19, and in this program, we are using properties of Java 8, such as lambda expressions and streams. A very important note in this assignment is that the weight of the edges will not be negative.

The given input is given through a text file, which will contain the following properties. The first line of the text file will contain two numeric values, separated by a space, in which the first one determines the total number of vertices (nodes) and the second one determines the total number of edges. Then, the second line provides the names of the nodes which will be separated by spaces. However, there is another possibility, in which the second line contains the label “use-indexes”, which nodes will be labelled by their index, starting from 0 to n -1, when n represents the number of vertices/nodes in the graph. Going to the third line, we will retrieve the index of the source node/vertex. The following *e* lines will be written in the format *source destination weight* that represents the directed edges between vertices and these will be separated by space. In this assignment there is also a bonus part, in which we will determine not only the shortest path, but also the longest path between vertices in both types of graphs, DAG and directed cyclic graph.

II. Solution Approach and Algorithm Outline

The program consists of 6 Java classes, which are Edge, Vertex, Graph, FileManagement, ShortestAndLongestPathCalculator, and Main. Moreover, it consists of two text files named graph.txt and graph1.txt. There are two text files to test for two different graphs, one for DAG, and one for directed cyclic graphs.

Firstly, the program creates an object of FileManagement class, and it calls the method readFile to read the content of the file, which follows the rule introduced in the first section. Therefore, in the first line, we save the number of total vertices and edges to two integer values, and also, we validate if the following inputs follow the requirements presented in the assignment. Therefore, the program checks the second line if the vertices are going to have a name for every vertex or if it will use indexes. If the program is using names, we are storing the name and index of the vertices in the HashMap<Integer, String>. In addition, in the third line, we retrieve the source vertex, saved in an integer value, which means that we will calculate the paths from that vertex to the other vertices. and the other lines represent the source, destination, and weight and are saved in a List<Edge>. So, for every line, we create an object of Edge, with three properties, source of type integer, destination of type integer, and weight of type double.

If the inputs meet the requirements, we create an instance of class Graph, with two parameters, which are listOfEdges, represented as an adjacency list, and allVertices, which represents the number of vertices in the graph. Then, the method isDAG is called in order to determine if the given graph is a DAG or not. A graph is said to be DAG if it does not contain any cycle, otherwise, it is a directed cyclic graph. For both cases, we implemented different algorithms and methods, to retrieve the shortest and longest path of the given graph.

If the graph is a DAG, we will use topological sorting to retrieve the shortest path and the longest path between the source node and other nodes in the graph. Topological sorting is a strategy used to linearly request the vertices of a coordinated diagram so that for each coordinated edge (u,

v), vertex u precedes vertex v in the ordering. This algorithm is only applied in DAGs, and the reason behind this is that this type of graph does not contain any cycle, and this indicates that the vertices may be linearly arranged, such that every directed edge point from a vertex earlier in the arrangement to a vertex later in the arrangement. This means, that each vertex is dependent solely on the vertices that are placed before it in the topological order. This method is appropriate for DAGs since it offers a reliable and meaningful ordering of the vertices based on their relationships. While directed cyclic graphs generate circular dependencies that prevent the establishment of linear order.

Moreover, the other option occurs when the graph is not DAG, and as a result, we used the Dijkstra algorithm, and the reason behind this is that it employs a greedy approach. It can be used for DAGs and non-DAGs, but topological sorting is more efficient for DAG graphs. In order to ensure that the distances are updated as effectively as possible, it searches the network in a greedy approach, always choosing the vertex with the smallest distance. In other words, it updates the distances between its adjacent vertices by relaxing the vertices with the smallest distance from the source vertex, which is chosen repeatedly. The technique makes sure that each iteration's distance to each vertex is the shortest one yet discovered. The major disadvantage is that this algorithm does not work for negative weights, but since our program uses only positive weights, this algorithm works properly. In cases where it contains a negative weight, the Bellman-Ford algorithm can be used. A significant point in this procedure is the attachment of the priority queue in the Dijkstra algorithm, which improves the efficiency of the procedure, which is reflected in its time complexity. The longest path in non-DAGs is considered to be an NP-hard problem, which means that there is no known polynomial-time algorithm that can solve it for all cases. However, we modified some properties in the Dijkstra algorithm to display the longest path.

III. Data Structures

The data structures used in this program are HashMap, Arrays, Stacks, Priority Queues, and Adjacency List. As we mentioned, we used Java programming language for implementing this coursework. So, the HashMap is an important data structure that implements the Map interface, and it provides the opportunity to store and retrieve-key value pairs. Moreover, it belongs to java.util.package, and also it is part of Java Collections Framework. This data structure is defined as key-value pair, in which the key is unique. In each key, we can store items since it uses an array of buckets to store zero or more entities. To access the entities, we just need to use the proper key. The HashMap in our case is to associate the vertex index with its name if the user wants to give a name for each vertex instead of its index. The reason behind this choice relies on the efficiency of data retrieval. This means that this type of data structure offers a constant time performance for the most basic operations such as get and put. Moreover, it retrieves the value in O(1), and the procedure of searching for a key is performed in O(1). The key of the HashMap stays unique since it does not allow duplicate keys, and if we try entering a duplicate key, the previous one will be deleted and replaced by the new key pair value. The last important benefit is that it is resized automatically, which ensures that the load factor is within a reasonable range.

The type of array used in this assignment are one-dimensional arrays and are implemented in our assignment more than one time. Some, usages of array in our program are for checking if a vertex is being visited or not, to put the vertex that the path goes from source to destination by finding its shortest or longest path, and more. The usage of this data structure relies on the fact that it is one of the easiest data structures, and it provides efficient storage and retrieval. As a result, we can access any element in the array by its index, and it is reflected in its time complexity. So, to retrieve an element from the array, we perform constant time.

Another data structure used is Stack. Stack is another data structure that follows the LIFO (Last-In-First-Out) principle, which means that the last element inserted into the stack will be the

first element to pop from the stack. The most used methods for stacks are push, pop, and peek which have the time complexity O(1). Respectively, it offers the possibility to add an element to the stack, remove and return the topmost element from the stack, and just return the topmost element without removing it. One of the most important uses of stacks in our program is during topological sorting. The function is used for finding the shortest and largest path for only directed acyclic graphs (DAGs). One of the benefits of using stack is that ensures proper ordering, which means that have an essential part of maintain the correct ordering of the vertices during the topological sorting process.

To move further, Priority Queue is another data structure used in our coursework, which is an abstract data type that is part of java.util package and it presents a queue in which every element is processed based on their priority. The element with the highest priority is dequeued from the priority queue. Moreover, the basic operations handled here are insertion, deletion, and peek, in which the insertion and deletion are done in O(logN), while peek is performed in constant time. The priority queue in our program is implemented inside the Dijkstra algorithm. The advantages of using this data structure are that it provides efficient priority-based processing, fast update of priorities, and also optimized the runtime complexity of this algorithm.

Finally, we chose the adjacency list for our implementation for two main reasons: first, it has a space complexity that is significantly better than the adjacency matrix at O(V + E), and second, it has a time complexity that is linear when compared to the quadratic time that the adjacency matrix offers. The list is a decently efficient data structure for our solution since, despite the matrix being quicker at checking for edges, it can only do so in linear time.

IV. Time Complexity

During the analyse of time complexity we will use two letters, which are V, and E. The letter V represents vertices, and E represents edges. In the following paragraphs, we are going to provide the time complexity of the most

The first function to determine the time complexity is readFile. It starts by opening the file that has a constant time O(1), and it just returns a stream of lines. Then the program processes each line of the text file, so it will do read E + 3 lines. In this scope, it deals with switch case operation, and each of the cases do constant time. Then, after the completion of the reading part, the file is closed with a time complexity of O(1). Overall, the time complexity of this function is O(E + 3), and since we have a constant we do not consider it, so it is O(E),

The other implementation that will be analysed are the methods that determine if the given graph is a directed acyclic graph or not. A graph is said to be a DAG if it does not contain any cycle. The methods that are used here are isCyclic and isDAG. Firstly, the method isCyclic will be analysed, and this function starts by initialization one index of visited and recursionStack arrays, that have a size equal to the number of vertices in the graph that is performed in constant time. In the second part of the method, it makes it possible by iterating over the edges of the current vertex and making recursive calls for unvisited vertices. Regarding the recursive calls, the method makes a recursive call to isCyclic with the destination vertex for each unvisited edge destination, but the number of recursive calls depends on the number of edges and the structure of the graph. Then, for each edge, the method checks if the destination vertex is visited or presented in the recursion stack with constant time. After the recursion calls, we have one initialization line and 1 return statement with constant time. To sum up, the time complexity of isCyclic is O(E). The next function is isDAG, in which the first two statements we have are the declaration of visited and recursionStack arrays, which have the size of the number of vertices in the graph, and each of them is initialized with a false value. The next step is looping over the vertices, so, it starts by checking if each of the

vertex is visited or not, and since visited is an array, it has constant time. Then, it continues by calling the method isCyclic if the vertex is not visited, and as discussed above it has a time complexity of O(E). If the isCyclic return true, it tells that the graph is not a DAG, otherwise it is a DAG. Overall it has a time complexity of O(V + E).

We continue by analysing topological sorting, which is used to find the shortest and largest path of a DAG. It starts by initializing the visited array of the specified index of the vertex to true and it is performed in constant time. Then, we have the iteration over the adjacent edges of the current vertex. The time complexity of iterative over the adjacent edges relies on the number of adjacent edges. Also, for each adjacent edge, the method checks if the destination vertex is visited or not, and it is done in constant time, then it makes a recursive call if the vertex is not visited. The number of recursive calls depends on the number of unvisited adjacent vertices, and the structure of the graph. Then, we continue by pushing the vertex to the ordering stack, which is performed in constant time. As a result, the time complexity is done in O(E).

Moreover, we have the shortestPathForDAG method, which starts by initialization of the list known as distance, and it is done V times. Moving on, we continue with the looping over the ordering stack, and it iterates over the order stack until it is empty, and it iterates at most V times. Popping the vertex from the stack is done in constant time, and continue by checking the distance of the current vertex that has a time complexity of O(1). Moreover, we update distances for adjacent vertices and the method iterates over the adjacent edges of the current vertex and updates the distance if a shorter path is found. The number of adjacent edges depends on the structure of the graph and can vary. Overall, the time complexity of the shortestPathForDAG method can be approximated as O(V + E).

Then, we have the longestPathForDAG method, which starts by initialization of the list known as distance, and it is done V times. Moving on, we continue with the looping over the ordering stack, and it iterates over the order stack until it is empty, and it iterates at most V times.

Popping the vertex from the stack is done in constant time, and continue by checking the distance of the current vertex that has a time complexity of O(1). Moreover, we update distances for adjacent vertices and the method iterates over the adjacent edges of the current vertex and updates the distance if a longer path is found. The number of adjacent edges depends on the structure of the graph and can vary. Overall, the time complexity of the longestPathForDAG method can be approximated as O(V + E).

Before calling one of these two methods, the method shortestAndLongestPathForDAG is called. It starts by initialization of the previous array in V times. Then, it calls the topologicalSorting method with time complexity O(E), and it continues, with the calling of one of the two methods mentioned above. So, the total time complexity is O(V + E).

Another important method is DijkstraAlgorithm which computes the shortest path. In this algorithm, we declared a priority queue that increases the efficiency of this algorithm. Then, it initializes the distance list with a time complexity of O(V). Then, we insert the initial vertex with a distance of 0, and the time complexity is O(logV), and it continues by polling the vertex from the priority queue, and each poll operation has a time complexity of O(logV). Another important step is edge relaxation, so for each edge in the adjacency list of the current vertex, the method performs edge relaxation by comparing the new distance to the destination vertex with the current distance and updating it if necessary. This operation is performed for each edge in the graph, and the total time complexity is O((V+E)logV).

The last method includes the time complexity of the longest path for non-DAG. We did a modification of the Dijkstra algorithm, mentioned above. It initialized the distance list, previous array, visited array, and priority queue. The code creates a priority queue and inserts the initial vertex with a distance of 0, with a time complexity of O(logV), and then it proceeds by polling the vertex from the priority queue, and it is performed in O(logV). Then, we have edge relaxation, and

this operation is performed for each edge in the graph. Overall it has time complexity O((V+E)logV).

V. Code Snippet

```java
public void readFile() {
    try (Stream<String> lines = Files.lines(Paths.get( first: ".",  ...more: "src", "graph.txt"))) {
        AtomicInteger lineCount = new AtomicInteger( initialValue: 1);
        lines.forEach(line -> {
            String[] parts = line.split( regex: " ");
            switch (lineCount.getAndIncrement()) {
                case 1 -> {
                    if (validateInputs(Integer.parseInt(parts[0]), Integer.parseInt(parts[1]))) {
                        setInputValidation(true);
                        setAllVertices(Integer.parseInt(parts[0]));
                    } else {
                        setInputValidation(false);
                    }
                }
                case 2 -> {
                    if (parts.length > 1) {
                        IntStream.range(0, getAllVertices()).forEach(i -> indexWithName.put(i, parts[i]));
                    } else if (parts[0].equals("use-indexes")) {
                        indexWithName = null;
                    }
                }
                case 3 -> setStartingNode(Integer.parseInt(parts[0]));
                default ->
                        listOfEdges.add(new Edge(Integer.parseInt(parts[0]), Integer.parseInt(parts[1]), Double.parse
            }
        });
```

```java
public static void DijkstraAlgorithm(Graph graph, int source, HashMap<Integer, String> indexNames) {
    PriorityQueue<Vertex> priorityQueue = new PriorityQueue<>(Comparator.comparingDouble(Vertex::getDistance));
    List<Double> distance = new ArrayList<>(Collections.nCopies(graph.getAllVertices(), Double.POSITIVE_INFINITY));
    int[] previous = new int[graph.getAllVertices()];
    distance.set(source, 0.0);
    previous[source] = -1;
    priorityQueue.offer(new Vertex(source,  distance: 0.0));
    while (!priorityQueue.isEmpty()) {
        Vertex currentVertex = priorityQueue.poll();
        int vertexIndexOfCurrentVertex = currentVertex.getVertexIndex();
        if (currentVertex.getDistance() > distance.get(vertexIndexOfCurrentVertex)) continue;
        for (Edge edge : graph.getAdjacencyList().get(vertexIndexOfCurrentVertex)) {
            int destinationVertex = edge.getDestination();
            double weight = edge.getWeight();
            double newDistance = distance.get(vertexIndexOfCurrentVertex) + weight;
            if (newDistance < distance.get(destinationVertex)) {
                distance.set(destinationVertex, newDistance);
                previous[destinationVertex] = vertexIndexOfCurrentVertex;
                priorityQueue.offer(new Vertex(destinationVertex, newDistance));
            }
        }
    }
```

```java
static void shortestAndLongestPathForDAG(Graph graph, int source, int allVertices,
                                         HashMap<Integer, String> indexNames, int shortestOrLongest) {
    int[] previous = new int[allVertices];
    previous[source] = -1;

    Stack<Integer> ordering = new Stack<>();
    boolean[] visited = new boolean[allVertices];

    IntStream.range(0, allVertices)
            .filter(i -> !visited[i])
            .forEach(i -> topologicalSorting(graph, i, visited, ordering));

    if(shortestOrLongest == 0){
        shortestPathForDAG(graph, source, indexNames, ordering, previous);
    }
    else if (shortestOrLongest == 1) {
        longestPathForDAG(graph, source, indexNames, ordering, previous);
    }
}
```

```java
private static void topologicalSorting(Graph graph, int vertexIndex, boolean[] visited, Stack<Integer> ordering) {
    visited[vertexIndex] = true;

    graph.getAdjacencyList().get(vertexIndex).stream()
            .filter(edge -> !visited[edge.getDestination()])
            .forEach(edge -> topologicalSorting(graph, edge.getDestination(), visited, ordering));

    ordering.push(vertexIndex);
}
```

```java
private static void printInformation(List<Double> distance, int source, int[] previous, int allVertices,
                                     HashMap<Integer, String> indexNames){
    if (indexNames == null)
        printInformationWithoutName(distance, source, previous, allVertices);
    else
        printInformationWithName(distance, source, previous, allVertices, indexNames);
}
```

VI. Project Management

The project was prepared by Sofi Çapi, and Markela Mykaj. Sofi was in charge of the file management, and graph classes. Moreover, she handled the shortest and longest path for directed acyclic graphs. On the other hand, Markela was part of the implementation of the shortest and longest path in directed cyclic graph.

VII. Conclusion

Finding the shortest path from the source node to each other node in the graph is made possible by the implementation of the shortest routes described above, which applies to both directed graphs with cycles and DAGs. As long as the weights supplied are non-negative (in the case of Dijkstra), the Dijkstra and topological sort algorithms, which are employed for non-DAGs and DAGs, respectively, demonstrate the efficiency and yield the right pathways. Also, the algorithm applied for the longest path demonstrates efficiency too.

VIII. References

Dasgupta, S., Papadimitriou, C., & Vazirani, U. (2006). *Algorithms.*