University of New York Tirana

Algorithms & Complexity

Coursework

Assignment: Paths in Graphs
Programming language: Java
Professor: Elton Ballhysa
Due: June 11th, 23:59
Worked: Meggy Pojani

Table of Contents:

SHORTEST PATH

Problem Statement

The input is a directed connected graph, which might contain loops. There are given the number of nodes and the starting node—the number of edges and their presence (its endpoints, direction and weight). The names of the nodes are mentioned or the nodes are named after indexes.
The outputs are the shortest paths from the starting node to the other nodes.
In this problem, there are two cases. One is when there are cycles and the other is when they are not.

Solution

To solve this problem we could use Dijkstra's algorithm, but in both cases (the cyclic and acyclic) they will have the time complexity O((e+n)logn). This time complexity is good, but we can do better for DAGs by modifying the DFS algorithm to keep track of paths while it iterates through all the nodes. We can change it further, since we do not know if the given graph has cycles, to detect them. Only if a cycle is detected we use Dijktra's algorithm, which has worse time complexity in comparison.

Main

(L21-42) We use Scanner to get the information in the form of a String. (L27) We create an object of the class Graph. (L31-39)We change the String to an integer and save it to the variable *n*. We do the same to the number of edges in variable *e*. For both, we check if they are within their required constraints and exit the program otherwise. (L42-60)The names of the variables are saved in an array called *names*. In case there is the keyword "use-indexes", we save in the array indexes as Strings. (L65-67) We save the starting Node's index into the integer variable *startingNodeIdx.* (L73-91) We take each line and save it into the String *data* and then we put that information into an array *temp.* The weights are checked if they

are within the necessary constraints or the program stops. Then the edge is added to the Graph *g*. (L95) The algorithm is run (found in class Graph).
The base of the classes Node, Edge, Path and Graph are taken from the Graphs lesson of the Data Structure course, using chapter 14.1 of the book (Weiss, 2010) and then have been modified to be used in the project.

Class Node

(L13-17) The nodes are created as objects which have some necessary attributes such as String *name*, List<Edge> *adj* (a list of edges which connect them to adjacent nodes), int *dist* (distance from starting node), Node *prev* (it's previous node) and boolean *seen* (which is used to determine if the node has been visited before or not). (L19-23) There is a constructor Node(String *_name*), which creates a new Node with only its name and all the other attributes are empty, infinity (greatest value an integer can take Integer.MAX_VALUE), null or false, accordingly. (L26-32) A *reset()* method which turns all the attributes (except the *name*) to their empty forms.
The adjacency List of this graph is formed from the Node HashMap and the *adj* List, in which each node contains its adjacent edges. The HashMap is used so that getting and adding nodes has constant time complexity.
The creation of new Nodes has **O(1)** time complexity.

Class Edge

(L11-13) Each object of the Edge class has two attributes: Node *dest* (destination Node of the edge) and int *weight* (the weight of the edge). (L16-19) A constructor Edge(Node *m*, int *w*) creates the edge.
The creation of new Edges has **O(1)** time complexity.

Class Path

(L12-13) Each object of the Path class has two attributes: Node *dest* (destination Node of the path) and int *weight* (the weight of the path). (L16-19) A constructor Path(Node *m*, int *w*) creates the path.

The creation of a new Path has **O(1)** time complexity.

Class Graph

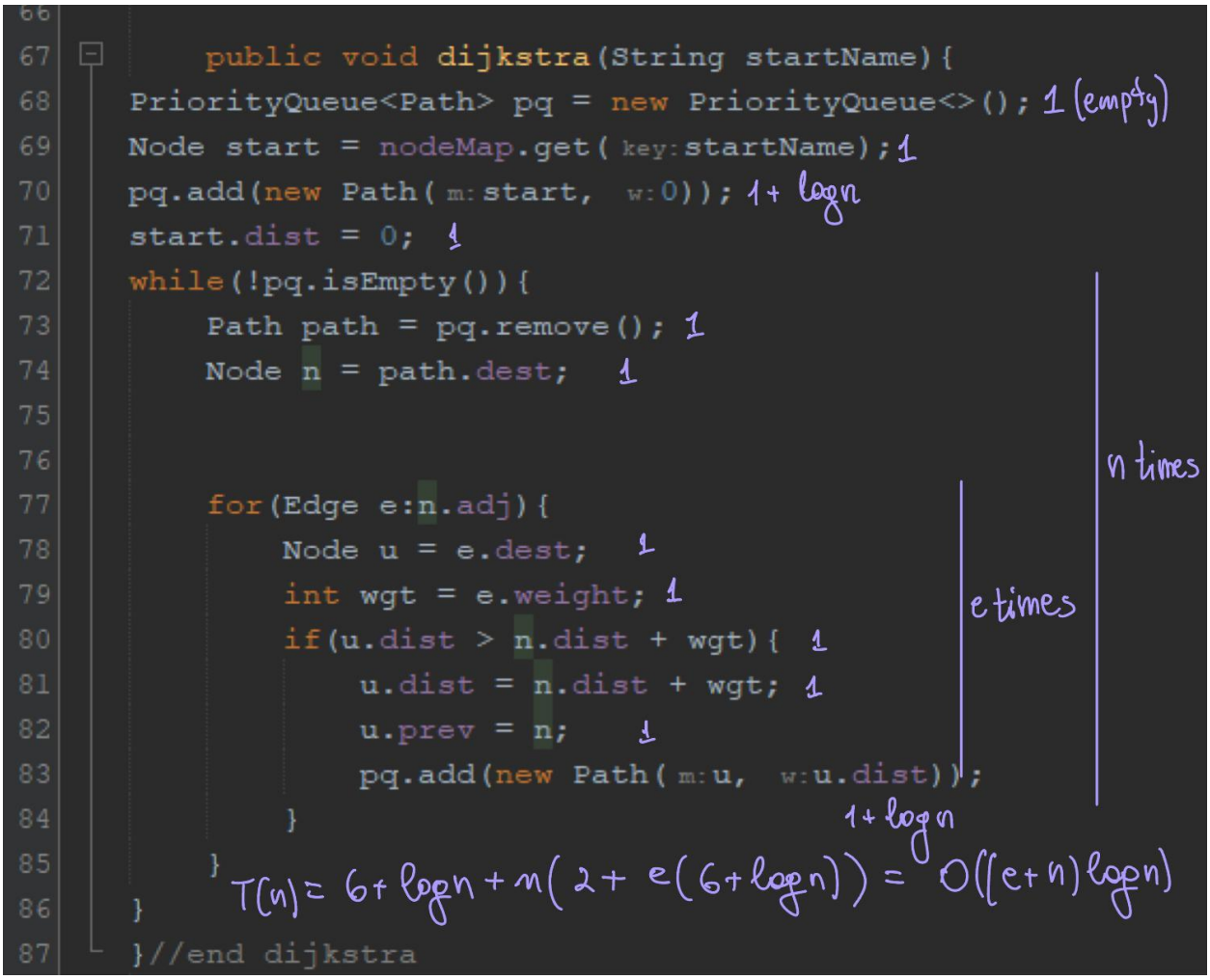
This is the most important class as it contains most of the information for the graph and the algorithms.
(L15) We use a HashMap *nodeMap* to save all the nodes and edges because the insertion and retrieval time complexity of a HashMap is **O(1)**. (L24-30) getNode(String *nodeName*) checks if the node with that name exists in the Graph (already has been entered) and if not creates a new node with the name *nodeName.* In any case, the node is returned. (L38-43) addEdge(String *sourceName*, String *destName*, int *weight*) method adds a new edge to the Graph. (L54-57) printPath(Node *destination*) is a method that uses a recursive algorithm to the names of all the ancestors (previous nodes) of the imputed Node. This is run a maximum of n times **O(n)**. (L58-61) *printAllPaths*(String[] *names*) for each node ir runs printPath method and in its entirety has T(n) = **O(n^2)**.

Dijkstra's Algorithm

Input: String *startName,* the name of the starting node.
We use the Dijkstra algorithm from the lecture slides:

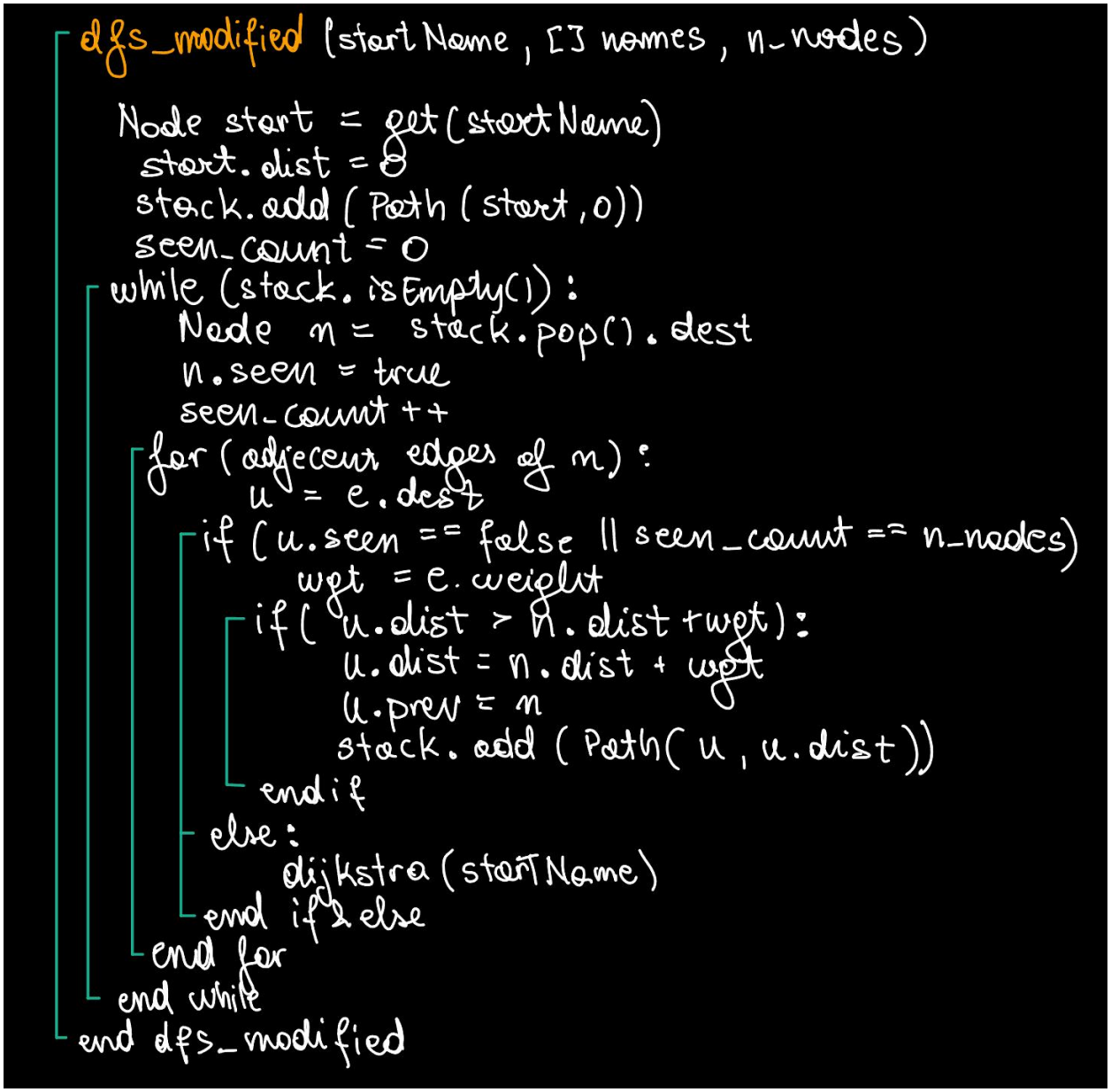
Algorithm 2 Dijkstra's Algorithm

```
1: procedure DIJKSTRA(G, l, s)
2:     for all (u) ∈ V do
3:         dist(u) = ∞
4:         prev(u) = nil
5:     end for
6:     dist(s) = 0
7:     H = make-queue(V)                          ▷ using dist values as keys
8:     while H ≠ ∅ do
9:         u = delete-min(H)                      ▷ process the closest node
10:        for each edge (u, v) ∈ E do
11:            if dist(u) + l(u, v) <dist(v) then
12:                dist(v) = dist(u) + l(u, v)    ▷ shorter path s → v found
13:                prev(v) = u
14:                decrease-key(H, v)
15:            end if
16:        end for
17:    end while
18: end procedure
```



The difference in input is that since the method is in the Graph class (and it is not static) the Graph is given. The input is only the name of starting node. We get the Node from the graph based on its name. We add that node's path with weight 0 to the created PriorityQueue *pq*. We also make that node's distance 0.
The while loop will continue until *pq* becomes empty.
For each edge of this node, the destination node is taken and if its dist is larger than the sum of the distance with the weight of the edge then that distance is replaced and the previous is made the node *n*. This node is then added to *pq*.
There is a small difference between the implementation and the lecture algorithm in that the Priority Queue is empty and we slowly add the nodes. We do not create it with all the nodes and then decrease-key(H,v).

Modified DFS algorithm



Inputs: starting node's name, the array with the node's names, number of nodes
We get the starting node in the object *start.* We add its path to the stack *st* and make its dist = 0. We initialise the *seen_count* to 0, for if exceeded the number of nodes then we can detect cycles.
The while loop continues as long as stack *st* is not empty. We take the node from the top of the stack, turn its *seen* status to true and increment *seen_count*.
The for loop continues for every adjacent edge of the node, which was popped from the stack. The destination node is taken from the edges and if its dist is larger

than the sum of the distance with the weight of the edge then that distance is replaced and the previous is made the node *n.* The new path is added to the stack. (Line 105) Makes sure there are no cycles. If we come across a previously seen node or have seen more nodes than they are, the algorithm is not equipped to deal with it so we request dijkstr().

```
89      public void dfs(String startName, String[] names, int num_n){
90          System.out.println( x: "Print shortest paths:");
91  Stack<Path> st = new Stack<Path>();          1
92  Node start = nodeMap.get( key: startName);   1
93  st.add(new Path( n: start,  w: 0));          2
94  start.dist = 0;                              1
95  int seen_count = 0; //use it so we don't backtract unecesarry edges in DAGs
96  while(!st.isEmpty()){  n times
97      Path path = st.pop();  1
98
99      Node n = path.dest;    1
100     n.seen = true;         1
101     seen_count ++;         1
102
103     for(Edge e:n.adj){  takes each edge once
104         Node u = e.dest;    1
105         if(u.seen == false || seen_count <= num_n){ 1
106         int wgt = e.weight;  1
107         if(u.dist > n.dist + wgt){ 1
108             u.dist = n.dist + wgt; 1
109             u.prev = n;  1
110             st.add(new Path( n: u,  w: u.dist)); 2
111              }}
112         else{
113             System.out.println( x: "Dijkstra in use:");
114             dijkstra(startName);     (n+e) logn
115             printAllPaths(names);
116
117             System.exit( status: 0);
118         }
119     }          T(n) = O(n+e)
120 }
121 printAllPaths(names);
122 }//end dfs
```

The Dijkstra algorithm is used only for graphs with cycles. In worst case scenario where a cycle is found in the last node of the previously thought DAG, the time complexity is still **O((n+e)logn)** because T(n) = O(n+e) + O((n+e)logn).

LONGEST PATH

Using the same project without a Dijkstra algorithm and instead of initialising all the dist to the infinite, we initialise them to *INTEGER.MIN_VALUE*. In dfs method, which now is called dfs_longest, we just look made *if(u.dist < n.dist + wgt)* instead of it being bigger. This way we will get the longest path instead of the shortest. This solves all the problems for the DAGs, but for the cyclic graph, we do not add anything to the stack if it has been *seen* before. This way we do not calculate the edges that form cycles, we assume they are not there.

References:

Weiss. (2010). *Data Structures & Problem Solving Using Java* (4th ed.) [Ebook]. Pearson Education.