ALGORITHM DESIGN AND ANALYSIS

LAB 3, 2015

Instructions

(1) Please submit a single file `Digraph.java` before the end of the lab session
(2) In each file, you MUST use the correct indentation, and comments.
(3) You may work in a group of no more than 3 people
(4) The total mark of your lab is 30. It is worth 5% of your final grade

Building a Digraph Toolbox The program `Digraph.java` implements a simple directed graph ADT using the **adjacency list** data structure. Your task is to complete this class.

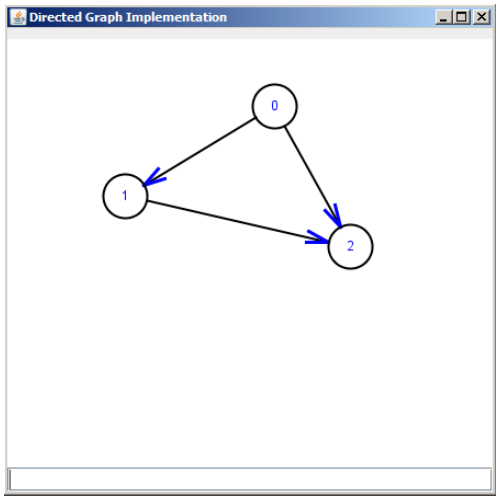

FIGURE 1. The Digraph ADT GUI

File to download:

- `Digraph.java`: This is the class you need to work with. You need to fill in the code as indicated in the file.

Data structures: The program uses the following main data structures:

- Each node is identified by an `Integer` label; no two nodes have the same label.
- `HashMap<Integer,List> data`: associates a node label with the list of out-neighbours of the node. This is the adjacency list of the digraph.
- `Node`: inner class handles the visualisation (GUI) of a node. This inner class specifies properties such as its position, and methods such as draw.
- `HashMap<Integer,Node>`: `nodeList` associates a node label with the `Node` object that represents this node.
- `Set<Integer> nodeSet`: stores the key set of `data`

Control: The program can be controlled in two ways: by mouse events or by commands from the text field located at the bottom of the frame; Detailed as follows:

- *Mouse Control*:

(1) To add a node: click on the white space in the panel. The newly added node will be labeled by the smallest available number, and be automatically *selected*.
(2) To move a node: drag and drop the node to any position within the panel
(3) To select a node: click on the node
(4) To add an edge: once a node is selected (showing in red), add an outgoing edge to the selected node by clicking the target node of this edge

- *Textfield Commands*:
  (1) `add node i`: Add a new node with label `i` (if `i` is not in the digraph yet)
  (2) `add edge i j`: Add a new edge from `i` to `j` (if both `i`, `j` are in the digraph and there is no edge from `i` to `j` yet)
  (3) `remove node i`: Remove the node `i` (if it is in the digraph)
  (4) `remove edge i j`: Remove the edge (`i`, `j`) (if it is in the digraph)
  (5) `print order`: Print the order of the digraph in the command line window
  (6) `print size`: Print the size of the digraph in the command line window
  (7) `print degree`: Print the in/out-degree of each node in the command line
  (8) `print list`: Print the adjacency list of the digraph in the command line
  (9) `print matrix`: Print the adjacency matrix of the digraph (see below for details) in the command line window
  (10) `transpose`: Compute the *transpose* of the current digraph and display it in the GUI. (see below for details)
  (11) `underlying`: Compute the *underlying graph* of the current digraph and display it in the GUI. (see below for details)
  (12) `clear`: Delete all nodes in the digraph
  (13) `dfs`: Performs depth first search traversal on the current graph starting from the smallest node, and print out the resulting DFS forest as a string.
  (14) `linearise`: Decides if the current graph contains a cycle. If it does not contain a cycle, print a linearisation of the graph.

Your tasks: You need to complete the following methods

(1) `void remove(int node)`:
This method removes `node`, if it is a node in the graph.
(2) `int indegree(int node)`:
This method computes and returns the in-degree of a given node `node`
(3) `void printMatrix()`:
This method prints out the adjacency matrix of the graph:
  (i) `HashMap labels`: associates each number between 0 and $n-1$ a unique node label
  (ii) `boolean adjMatrix`: an $n \times n$ matrix storing the adjacency matrix where the `i`th row/column corresponds to the node `labels.get(i)`
The method then prints out the adjacency matrix. To the left and on top of the matrix, the method also prints out the node label which corresponds to each row and column. For example, for the graph shown in Figure 1 the printed adjacency matrix should be

```
-------------------------
        0       1       2
0       0       1       1
1       0       0       1
2       0       0       0
-------------------------
```

(4) `void transpose()`:
The **transpose** of a digraph is defined as the digraph with the same set of nodes, but the direction of all edges are reversed. For example, the transpose of the digraph in Fig. 1 is shown in Figure 2. This method converts the current digraph to its transpose.
(5) `void dfs()`

This method performs depth first search traversal on the current graph starting from the node with the smallest index. The resulting DFS forest is represented as a sequence of brackets (as explained in the slides) and is printed in the command line. Figure 3 illustrates an example of running the `dfs` operation on a graph.

(6) `void linearise()`
This method checks whether the graph is linearisable. If it is not linearisable, the method prints out a message saying "The graph contains a cycle". If it is linearisable, the method applies DFS algorithm to print out a linearisation.

File to submit: `Digraph.java`

Marking schedule:

Method	Marks
`remove`	4
`indegree`	4
`printMatrix`	6
`transpose`	6
`dfs`	6
`linearise`	4
Total	30

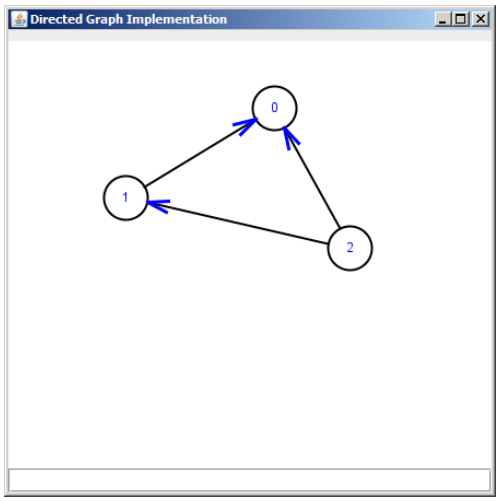

FIGURE 2. The transpose of the digraph shown in Fig. 1

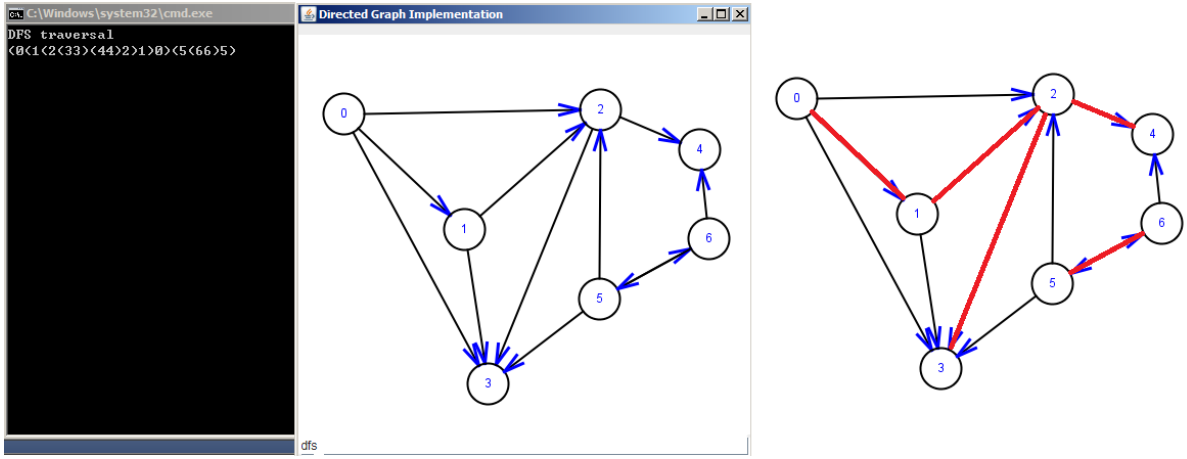

FIGURE 3. Running DFS traversal on the graph starting from the smallest node. The resulting DFS forest is printed on the command line window (left). The current graph is depicted on the GUI window (centre). The DFS forest is highlighted (right).