Practical 6

COS212

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science

Deadline: 12/04/2023 at 23:59

Marks: $\lfloor \pi * 100 \rfloor$

1 General instructions:

- This assignment should be completed individually; no group effort is allowed.
- Be ready to upload your assignment well before the deadline, as no extension will be granted.
- You may not import any of Java's built-in data structures. Doing so will result in a mark of zero. You may only make use of native arrays where applicable. If you require additional data structures, you must implement them yourself.
- If your code does not compile, you will be awarded a zero mark. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- All submissions will be checked for plagiarism.
- Read the entire assignment before you start coding.
- You will be afforded five upload opportunities.
- Make sure your IDE did not create any packages or import any libraries which are not allowed.

2 Plagiarism

The Department of Computer Science considers plagiarism a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with permission) and copying material from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm. **If you have any questions regarding this, please ask one of the lecturers to avoid misunderstanding.** Also, note that the OOP principle of code reuse does not mean you should copy and adapt code to suit your solution.

3 Outcomes

Upon completing this assignment, you will have implemented the following:

- a directed graph using the adjacency matrix representation.
- graph traversal algorithms.
- graph cycle detection.
- a graph connectivity algorithm.

4 Background

Graphs are data structures used to store data which is neither hierarchical nor ordered. It is used when the data has links which are not easily represented by other data structures. Graphs are usually implemented using Vertex and Edge classes and then storing these in arrays. This is the easiest and most natural representation for graphs, but this Practical will use the adjacency matrix representation. There are some algorithms which are designed with this representation in mind, but there are some algorithms which need minor modifications to work.

5 Tasks

5.1 Important information

Please note the following extra instructions, as you will lose marks if you don't follow all of them.

- You must use the adjacency matrix representation for this practical. This will be enforced by allowing only the given class for the Practical.
- Make sure you only submit Graph.java and no other files. Also, ensure you only have one class inside Graph.java and that there are no subclasses.
- You cannot import any data structures and may only use the default Java variable types. **It is important to note that you may not change any of the functions or variables given to you.**
- The UML and functions given are the bare minimum; feel free to add any functions and variables needed to complete the functions given.
- *Tip: Some algorithms require a Queue or a Stack, you can simulate this using an array.*
- Use Section 6 in combination with the given Main and output. Make use of text comparison techniques to ensure that your output matches exactly.

5.2 General instructions

You are only allowed to use one class for this Practical. The UML for the Graph class is given in Figure 1. You are not allowed to change any of the members or functions for this class. You are allowed to add any functions or members which you see fit. You are not allowed to add any classes, and you are not allowed to import any libraries other than the ones given.

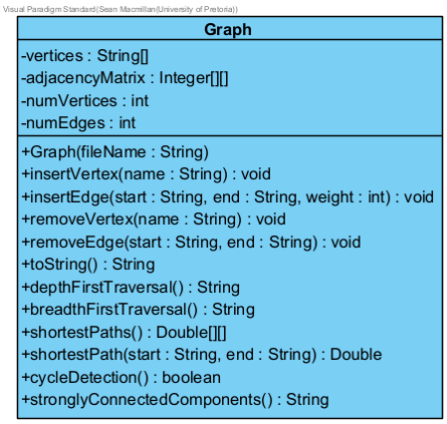

Figure 1: UML diagram

5.3 Graph

- Members
  - vertices:String[]
    - This is an array of Strings used to store the names of the vertices in the graph.
    - The array is NOT sorted, and the order will always be the order in which nodes were added, either from the text file or from the insertVertex function.
    - This array will always be traversed front to back. Thus in later functions, when multiple vertices need to be visited, this will be done in the order of this array.
    - If the graph is empty, this will be an array of size 0.
  - adjacencyMatrix:Integer[][]
    - This an 2D array of Integers. It is used to store the adjacency matrix used to represent the graph.
    - A value of 0 indicates that there are no links between two nodes. Any other value means there is an edge between those two nodes.
    - The array will always have numVertices rows and numVertices columns.
    - The array can be read as follows:
      - A row r shows the links from the r'th vertex in Vertices. Thus an index of 1 means that we are dealing with the second vertex in Vertices.
      - A column c then shows the weight from the r'th vertex to the c'th vertex. Thus an index of 2 means we are dealing with the third vertex in Vertices.
      - Thus, the entry [1][2] is the weight from the second vertex to the third vertex.
    - If the graph is empty, this will be a 2D array of size 0x0.
  - numVertices:int
    - This is the number of vertices in this graph.
  - numEdges:int
    - This is the number of edges in this graph.
- Functions
  - Graph(fileName:String)
    - This is the constructor for the Graph class.
    - If the passed in fileName is an empty string, you should initialise the graph to an empty graph.
    - If the passed in fileName is not empty, you should use this to initialise the graph.
    - You may assume that the file will always exist when the fileName is not empty.
    - Use the example in Section 6 to see the file's structure.
    - A brief overview of the file structure is shown below:

```
numVertices
Vertices Array (separated by spaces)
row 1 of the adjacency matrix (separated by spaces)
row 2 of the adjacency matrix (separated by spaces)
...
...
row numVertices of the adjacency matrix (separated by spaces)
```

- insertVertex(name:String):void
  - This function adds a vertex to the graph.
  - The vertex should be added to the end of the vertices array.
  - The adjacency matrix should be updated, assuming there are no edges from this vertex to the other vertices.
  - You may assume that duplicates won't be inserted.
- insertEdge(start:String,end:String,weight:int):void
  - This function adds an edge to the graph.
  - If either of the Strings is not found in the graph or the weight is 0, do nothing.
  - The edge is a directed edge from the Vertex with the name start to the Vertex with the name end. The weight of this edge is the passed-in weight parameter.
  - If there is already an edge between those nodes change the weight of the edge to the new value.
- removeVertex(name:String):void
  - This function removes a Vertex from the graph.
  - If the Vertex is not found, do nothing.
  - The vertices and adjacencyMatrix arrays should be updated to reflect this.
- removeEdge(start:String,end:String):void
  - If either start or end is not found inside the graph, do nothing.
  - Set the weight from Vertex start to Vertex end to 0.
- toString():String
  - This function returns a String representation of this graph.
  - If the graph is empty, return the String "(0,0)"
  - Use the example in Section 6 to see the required format.
  - An overview is shown below:

```
(numVertices,numEdges) \t vertex1 \t ... vertexN
vertex1 \t weight1-1 \t ...weight1-N
...
...
vertexN \t weightN-1 \t weightN-N
```

- depthFirstTraversal():String
  - This returns a depth-first traversal of the graph.
  - The Vertex's name should be used and not the index in the array.
  - Every name should be between square brackets.
  - These square brackets are then appended to each other and then returned (without a new line)
  - If the graph is empty, return an empty string.
- breadthFirstTraversal():String
  - This returns a breadth-first traversal of the graph.
  - The Vertex's name should be used and not the index in the array.
  - Every name should be between square brackets.
  - These square brackets are then appended to each other and then returned (without a new line)
  - If the graph is empty, return an empty string.
- shortestPaths():Double[][]
  - This returns the All to All shortest path algorithm result.
  - Double.POSITIVE_INFINITY should be used for unreachable nodes.
  - If the graph is empty return an array of size 0x0.
- shortestPath(start:String,end:String):Double
  - This returns the distance for the shortest path between the start and end vertex.
  - If either of the nodes is not in the graph, return null.
- cycleDetection():boolean
  - Return true if there is a cycle in the graph.
  - Return false if there are no cycles in the graph.
- stronglyConnectedComponents():String
  - This returns a String showing the strongly connected components of the graph.
  - This algorithm displays subgraphs where all nodes can reach each other within every subgraph.
  - Vertices should be displayed using the name of the vertex inside square brackets.
  - Every subgraph should be displayed on its own line (the last line should also end a newline).
  - The nodes in each subgraph should be displayed by concatenating the strings without spaces.
  - If the graph is empty, return an empty string.

6 Example

This section should be used together with the provided files. The given Main tests all the functions that return Strings, which should ensure that your formatting is correct. "Graph.txt" is the textfile used to create the graph for testing. This graph is shown in Figure 2. Use this picture to test the logical correctness of the rest of your functions.

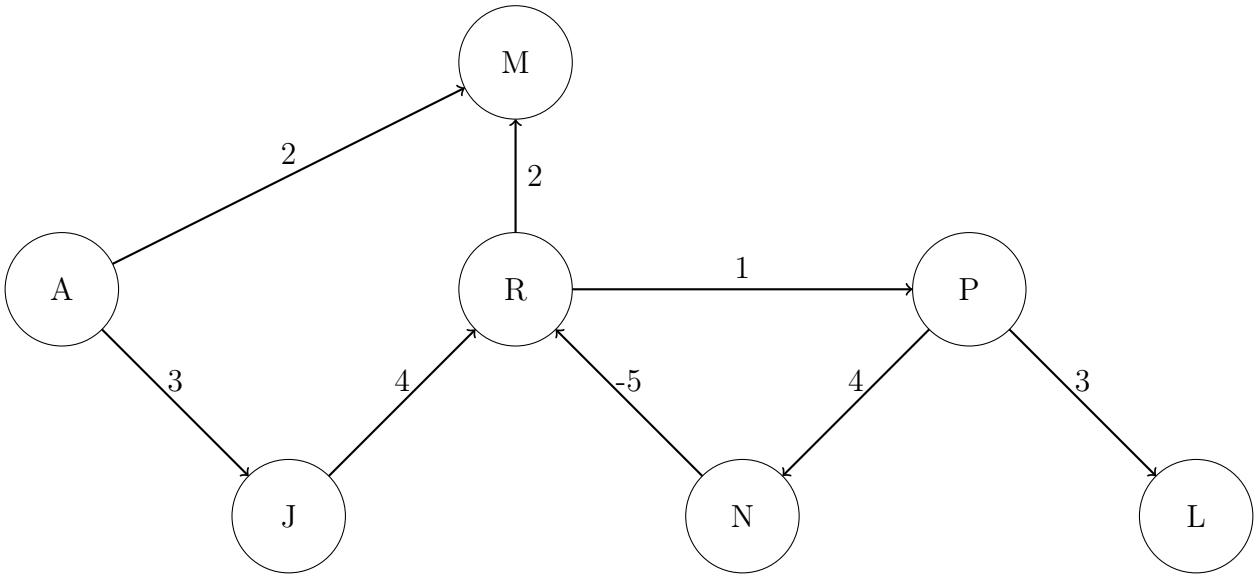

Figure 2: Example Graph

The vertices array is shown below. This was read from the textFile and will be used to determine the traversal order for the practical.

```
[A,J,L,M,N,P,R]
```

The adjacency matrix is shown below, also read from the textfile.

$$\begin{vmatrix} 0 & 3 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -5 \\ 0 & 0 & 3 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 1 & 0 \end{vmatrix}$$

7 Submission instructions

Do not submit any custom function classes.
Your code should be able to be compiled and run with the following commands:
javac Main.java Graph.java
java Main

Once you are satisfied that everything is working, you must create an archive of all your Java code into one archive called uXXXXXXXX.{tar.gz/tar/zip} where X is your student number. Submit your code for marking under the appropriate link before the deadline. **Make sure your archive consists of only java files and no folders or subfolders.**

Please ensure that you thoroughly test your code before submitting it. Just because your code runs with local testing does not mean that your code works for every situation. **DO NOT USE FITCH FORK AS A DEBUGGER**

8 Submission checklist

Submit the following files:

- Graph.java
- Make sure no other files are uploaded

9 Allowed imports

The following imports are allowed:

- java.io.File
- java.io.FileNotFoundException
- java.util.Scanner