JOHANNES KEPLER
UNIVERSITÄT LINZ

Algorithms and Data Structures 2
Winter term 2019
Stefan Grünberger, Dari Trendafilov

Assignment 08

Deadline: **Thu. 2.1.2020, 23:59**
Submission via: **www.pervasive.jku.at/Teaching/**

Elaboration time

*Remember the time you need for the elaboration of this assignment and document it in the file **time.txt** according to the structure illustrated in the right box. Please do not pack this file into an archive, but upload it as a **separate file**.*

```
#Student ID
k12345678
#Assignment number
07
#Time in minutes
190
```

Graphs

1. Edge list and adjacency matrix 12 points

Implement a **directed** and **weighted** graph **without loop** using an **edge list**. The vertices of the graph are defined in the interface `MyVertex`, the edges are defined in `MyEdge`. The graph itself is defined by the class `Graph`.

```
public interface MyVertex {
  // returns a vertex in form of a string.
  public String toString();
}
```

```
public class MyEdge {
  public int in,out;              // indices of the vertices
  public int weight;              // weight of the edge
}
```

```
import java.util.Arrays;

public class Graph {

  protected MyVertex vertices[]; // vertex array
  protected MyEdge edges[];      // edge array

  // Creates an empty graph
  public Graph() {
    vertices = new MyVertex[1];
    edges = new MyEdge[0];

    ...
  }


  // increase the size of edge and vertex array
  private void doubleArraySize() {
    int arraySize = vertices.length;
    vertices = Arrays.copyOf(vertices, arraySize*2);
    edges = Arrays.copyOf(edges, arraySize*2 * (arraySize*2-1));
  }

  // Returns the number of vertices in the graph.
  public int getNumberOfVertices() { ... }

  // Returns the number of edges in the graph.
  public int getNumberOfEdges() { ... }

  // Returns an array of length getNumberOfVertices() with the inserted vertices.
  public MyVertex[] getVertices() { ... }

  // Returns an array of length getNumberOfEdges() with the inserted edges.
  public MyVertex[] getEdges() { ... }


  // Insert a new vertex v into the graph and return its index in the vertex array.
  // If the vertex array is already full, then the method doubleArraySize() shall be called
  // before inserting.
  // Null elements are not allowed (IllegalArgumentException).
  public int insertVertex(MyVertex v)
    throws IllegalArgumentException { ... }
  // Returns true if there is an edge between index v1 and v2, otherwise false.
  // In case of unknown or identical vertex indices throw an IllegalArgumentException.
  public boolean hasEdge (int v1, int v2) throws IllegalArgumentException { ... }

  // Inserts an edge between vertices with v1 and v2. False is returned if the edge already exists,
  // true otherwise. An IllegalArgumentException shall be thrown if the vertex indices are unknown or
  // if v1 == v2 (loop).
  public boolean insertEdge(int v1, int v2, int weight)
    throws IllegalArgumentException { ... }

  // Returns an NxN adjacency matrix for the graph, where N = getNumVertices().
  // The matrix contains 1 if there is an edge at the index position, otherwise 0.
  public int[][] getAdjacencyMatrix() { ... }

  // Returns an array of vertices which are adjacent to the vertex with index v.
  // If the vertex index v is unknown an IllegalArgumentException shall be thrown.
  public MyVertex[] getAdjacentVertices(int v)
    throws IllegalArgumentException { ... }

}
```

2. DFS traversal 12 points

Extend the class `Graph` and implement the **DFS (Depth First Search)** algorithm and the following public methods, that use the DFS. It should be used to check if the graph is connected (`isConnected`), the number of components it consists of (`getNumberOfComponents`) and if it contains cycles (`isCyclic`).
Additionally, implement a method that outputs the vertices of the respective components line by line (`printComponents`).

As these calculations are more sophisticated for directed graphs, **temporarily convert the directed graph into an undirected graph** for the duration of the function call. This can be achieved by inserting directed edges in the opposite direction of the existing edges.
Consider: Efficient removal of the temporary edges after the calculation of the result!

```
  // Returns true if the graph is connected, otherwise false.
  // For the duration of the calculation temporarily convert the directed graph to an undirected graph.
  public boolean isConnected() { ... }

  // Returns the number of all weak components
  // For the duration of the calculation temporarily convert the directed graph to an undirected graph.
  public int getNumberOfComponents() { ... }

  // Prints the vertices of all components (one line per component).
  // For the duration of the calculation temporarily convert the directed graph to an undirected graph.
  public void printComponents() { ... }

  // Returns true if the graphs contains cycles, otherwise false.
  // For the duration of the calculation temporarily convert the directed graph to an undirected graph.
  public boolean isCyclic() { ... }
```