Practical Work nr. 2

Problem statement

3. Write a program that finds the connected components of an undirected graph using a depth-first traversal of the graph.

Feature list

F1.	Add a vertex
F2.	Remove a vertex
F3.	Add an edge
F4.	Remove an edge
F5.	Find the Connected Components
F6.	Clear screen
F7.	Exit

Program format

The graph will be read from a text file having the following format:

- On the first line, **n** and **m**, **n** being the number of vertices and **m** the number of edges in the graph
- On each of the following **m** lines there are three numbers: **x**, **y**, describing an edge of an undirected graph

Menu:

> Select a command:

1 - Add VERTEX

2 - Remove VERTEX

3 - Add EDGE

4 - Remove EDGE

5 - Find the CONNECTED COMPONENTS

h - Help

c - Clear screen

0 - Exit

File input example:

test0.txt

5 6
0 0
0 1
1 2
2 1
1 3
2 3

Running scenario

	`> Select a command:`
	`  1 - Add VERTEX`
	`  2 - Remove VERTEX`
	`  3 - Add EDGE`
	`  4 - Remove EDGE`
	`  5 - Find the CONNECTED COMPONENTS`
	`  h - Help`
	`  c - Clear screen`
	`  0 - Exit`

5	

	`> CONNECTED COMPONENTS:`
	`1.  0  1  2  3`
	`2.  4`

1	
	`> Vertex:`
5	
	`> The vertex was successfully added.`
3	
	`> First vertex:`
4	

	> Second vertex:
5	
	> The edge was successfully added.
5	
	> CONNECTED COMPONENTS: 1. 0 1 2 3 2. 4 5
4	
	> First vertex:
2	
	> Second vertex:
3	
	> The edge was successfully removed.
4	
	> First vertex:
1	
	> Second vertex:
2	
	> The edge was successfully removed.
5	
	> CONNECTED COMPONENTS: 1. 0 1 3 2. 2 3. 4 5
4	
	> First vertex:
2	
	> Second vertex:
5	
	> Edge does not exist.

0	
	`> Exiting...`

Specification

We shall define a class named Graph representing a directed graph.

Graph

- a class that creates a graph using a dictionary with inbound and outbound edges for every vertex and also a dictionary with the cost for every edge.
- initially, the graph is empty; the user may choose to load a graph from a text file or input the data manually; each line of the text file contains 2 vertices and the cost of their edge.

The ***Graph class*** will provide the following methods:

1. **addVertex(v)** : adds a new vertex, v, to the graph
2. ***removeVertex(v)*** : removes vertex v from the graph (if such a vertex exists) and all the edges that vertex has
3. **addEdge(x, y)** : adds an edge from x to y to the graph
4. **removeEdge(x, y)** : removes the edge from x to y from the graph (if the edge exists)
5. **dfs(v, k)**: builds a dictionary of connected components

Implementation

- The vertices dictionary contains as key a vertex v and as its value a list of elements u such that there is an edge from v to u.
- The visited list contains the list of nodes in the graph as indexes and as values: 0 if the vertex has not been visited or 1 if the vertex has been visited.
- Initially the dictionary is empty. When a vertex is added, a key with the specified value is created in the vertecies dictionary along with an empty list. When an edge (x, y) is added, y is added to the list of elements of the key x and x is added to the list of elements of the key x.

The implementation for adding a vertex

```
def addVertex(self, v):
 # adds a vertex to the graph
       # returns 0 if the addition was successful
       # return -1 otherwise

       if v in self.vertices:
           return -1

       self.vertices[v] = []

       return 0
```

The implementation for removing a vertex

```
  def removeVertex(self, v):
       # remove a vertex from the graph
       # returns 0 if the removal was successful
       # return -1 otherwise

       if v not in self.vertices:
           return -1

       for node in self.vertices[v]:
           self.vertices[node].remove(v)

       self.vertices.pop(v)

       return 0
```

The implementation for adding an edge

```
def addEdge(self, x, y, c):
       # adds an edge to the graph
       # returns 0 if the addition was successful
       # return -1 otherwise

       if x in self.vertices:
           if y in self.vertices[x]:
               return -1

       if y not in self.vertices[x]:
           self.vertices[x].append(y)
       if x not in self.vertices[y]:
           self.vertices[y].append(x)

       return 0
```

The implementation for removing an edge

```
def removeEdge(self, x, y):
      # remove an edge from the graph
      # returns 0 if the removeal was successful
      # return -1 otherwise

      if x not in self.vertices:
          return -1

      if y not in self.vertices[x]:
          return -1

      if y in self.vertices[x]:
          self.vertices[x].remove(y)
      if x in self.vertices[y]:
          self.vertices[y].remove(x)

      return 0
```