PWr Week 11

Data Structures and Algorithms

Laboratory – **List 10**

In above tasks vertices a numbered from 0. Any graph is without loops (a loop is an edge form a vertex to this same vertex).

1. Implement graph representation as an adjacency matrix. Do following functions:
   a. **`void`** `loadGraph(Graph &g,` **`int`** `n,` **`double`** `m)` – which loads from a console a graph with `n` vertices and `m` edges. The format is presented below.
   b. **`void`** `insertEdge(Graph &g,` **`int`** `u,` **`int`** `v,` **`double`** `weight)` – insert vertex from `u` to `v` of the weight equal `weight`. If the edge already exists, change the weight to the new value.
   c. **`bool`** `findEdge(Graph &g,` **`int`** `u,` **`int`** `v,` **`double`** `&weight)` – find an edge from `u` to `v` and under variable `weight` return its weight. Return information if the edge exists.
   d. **`void`** `showAsMatrix(Graph &g)` – print on console the graph as an matrix. The format of output take from an example.
   e. **`void`** `showAsArrayOfLists(Graph &g)` – print on console the graph as an array of lists. The format of output take from an example.
2. Implement **another** graph representation as an array of adjacency lists, vertices in every lists have to be sorted depending on vertices number. Do the same functions like for task 1 for the second representation.

Format of input lines: m lines of the format:

```
<startingVertex> <endingVertex> <weightOfEdge>
```

e.a:

```
2 4 3.5
```

Means an edge form 2 to 4 of the weight 3.5. Vertices are numbered from 0.

For **10 points** present solutions for this list till **Week 12**.
For 8 **points** present solutions for this list till **Week 13**.
For 5 **points** present solutions for this list till **Week 14**.
After Week 15 the list is closed.

Appendix 1

The solution will be automated tested with tests from console of presented below format. The test assumes, that there are up to X different graphs, which there are created as the first operation in the test. Each graph will be loaded from input stream.

If a line starts from '`#`' sign, the line have to be ignored.
In any other case, your program should print an exclamation mark and write (copy) introduced a line and then, depending on the command follow the correct procedure / function.

If a line has a format:
`GO` *`X`*
your program has to create *n* graphs (without initialization). The graphs are numbered from 0 like an array of lists. Default current graph is a graph with number 0. This operation will be called once as the first command.

If a line has a format:
`CH` *`n`*
your program has to choose a graph of a number `n`, and all next functions will operate on this graph. There is *n>=0* and *n<X*.

If a line has a format:
`LG n m`
your program has to call `loadGraph(g, n, m)` for current graph g. For any graph this operation will be called once, before using the graph. **The next `m` lines will present the information about edges.**

If a line has a format:
`IE` *`u v w`*
your program has to call `insertEdge(g,u,v,w )` for current graph `g`.

If a line has a format:
`FE u v`
your program has to call `findEdge(g,u,v,w)` for current graph `g`, and if the function return **`true`**, write on the output returned value `w`. Otherwise write "false" with new line character.

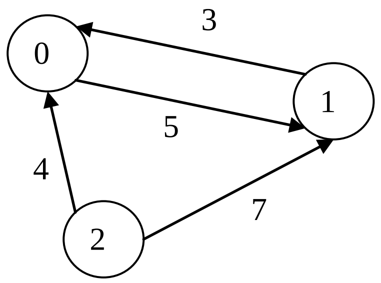
If a line has a format:
`SM`
your program has to call `showAsMatrix(g)` for current graph `g`.

If a line has a format:
`SA`
your program has to call `showAsArrayOfLists(g)` for current graph `g`.

If a line has a format:
`HA`

your program has to end the execution, writing as the last line “END OF EXECUTION”. Every test ends with this line.

A graph from example test:



For example for input test:

```
GO 2
LG 3 4
0 1 5
2 0 4
1 0 3
2 1 7
FE 0 2
FE 2 0
SM
FE 1 2
SA
HA
```

The output have to be:

```
START
!GO 2
!LG 3 4
!FE 0 2
false
!FE 2 0
4
!SM
0,5,-,
3,0,-,
4,7,0,
!FE 1 2
false
!SA
0:1(5),
1:0(3),
2:0(4),1(7),
!HA
END OF EXECUTION
```