Țibu Cristian
group 917
7th april 2023

Graph algorithms
practical work no. 1
implementation of a directed graph

Working in Python, we define a class called "**DirectedGraph**". Its ***__init__*** holds three dictionaries: *(in graph.py)*

```
class DirectedGraph:
    def __init__(self):
        self.nodesIn = {}
        self.nodesOut = {}
        self.edges = {}
```

For the first dictionary, "nodesIn", the keys are all the nodes of the given graph, while their values are the inbound neighbors.
*e.g: {"**x**": [**y**, **z**]} is the "nodesIn" for a graph which has edges going from **y** to **x**, and from **z** to **x**.*

For the second dictionary, "nodesOut", the keys are all the nodes of the given graph, while their values are their outbound neighbors.
*e.g: {"**x**": [**y**, **z**]} is the "nodesOut" for a graph which has edges going from x to y, and from x to z.*

For the third dictionary, "edges", the keys are tuples which represent edges, while their values are the weights of said edges.
*e.g: {(**x**,**y**): **w**} is the "edges" for a graph which has one single edge, that being one going fron node **x** to node **y**, with a cost of **w**.*

The "**DirectedGraph**" class provides multiple methods. Some of them are:

countNodes(), which returns the number of nodes,
countEdges(), which returns the number of edges,
checkNode(*node*), which returns True if the given node exists and False otherwise,
checkEdge(*node1, node2*), which returns True if the given edge (namely, the edge going from ***node1*** to ***node2***) exists, and False otherwise,
getWeight(*node1, node2*), which returns the weight of the given edge,
getInDegree(*node*), which returns the indegree of the given node,
getOutDegree(*node*), which returns the outdegree of the given node,
__str__(), which returns a string representing the entire graph, whose lines are of the form **<x --> y, weight w>**, considering an edge going from ***x*** to ***y*** with a weight of ***w***.

Some other, more complex methods of this class are:

addNode(*node*), which adds the given node to the graph. It does so by creating an empty list to both "nodesIn" and "nodesOut." If the node already exists within the graph, returns False; returns True otherwise.

removeNode(*node*), which removes the given node from the graph. It does so by removing it from both dictionaries mentioned above, as well as removing any "trace" it'd leave behind (i.e edges involving it, or it appearing in said dictionaries, being referred to as neighbor by other nodes)

addEdge(*node1, node2, weight*), which adds an edge to the graph, going from `***node1***` to `***node2***` and with a weight of `***weight***`. The tuple ***(node1, node2)*** is added to the dictionary "edges", with a value of `***weight***`.
On success, returns **True**.
On failure, returns a negative value; returns **-1** if `***node1***` does not exist; returns **-2** if `***node2***` does not exist; returns **-3** if the edge already exists.

removeEdge(*node1, node2*), which removes the edge described by `***node1***` and `***node2***`. The nodes will no longer be neighbors within the first two dictionaries, and the tuple ***(node1, node2)*** will be deleted from the third dictionary.
On success, returns **True**.
On failure, returns a negative value; returns **-1** if `***node1***` does not exist; returns **-2** if `node2` does not exist; returns **-3** if the edge does not exist.

parseNodes(), which returns an iterable (a list), containing the nodes of the graph. This list is sorted.

parseIn(*node*), which returns an iterable (a list), containing the inbound neighbors of `***node***`. This list is sorted.

parseOut(*node*), which returns an iterable (a list), containing the outbound neighbors of `***node***`. This list is sorted.

Additionally, service.py contains the following functions:

readGraph(*graph: DirectedGraph, filename: str*), which reads into `graph` from the file given by `filename`. The first line must contain the number of nodes, followed by a whitespace, and then the number of edges. The rest of the lines represent edges: the first number is the first node, the second number is the second node, and the third number is the weight of the edge.
This function does not return anything.

writeGraph(*graph: DirectedGraph, filename: str*), which writes into `filename` the given graph. The formatting is the same as in the function above.
This function does not return anything

createRandomGraph(*noNodes: int, noEdges: int*), which returns a graph of the class **DirectedGraph**, with randomly generated nodes and edges. The number of nodes is specified by `***noNodes***`, and the number of edges by `***noEdges***`. This function is guaranteed to return a graph with said number of nodes and edges.

All of the above is passed through a ***Service*** layer, which is then passed into an ***UI*** layer.
Additionally, the functionality of copying is provided:
`self._copies = []` will store, within the **Service** class, copies of the graph. The copying is done by **makeCopy()**, which will append into "_copies" a copy of the current graph. Reverting/changing to such a copy is done by **changeGraphToCopy()**, which will change the current graph to the last copy made, relative to the moment of calling.

Everything is wrapped up into a console-based menu, through which the user can perform all of the required operations for this practical work/laboratory.