Assignment 1 – deadline: lab 2 (week 3-4)

Implement an abstract data type Graph. Notes:

- The language used for implementation is Python 3.
- The graph must be a simple **directed** graph, meaning there are no self loops or multiple edges with the same endpoints and you have to consider orientations of the edges between the vertices.
- Vertices are specified as integers, from 0 to *n*-1, where *n* is the total number of vertices.
- Edges are either specified by an edge_id (i.e. you give a name to each edge) or by the pair of their endpoints (i.e. a pair formed by the initial_vertex and the terminal_vertex). The edge_id can be an int, a str or a custom class made by you.
- You are allowed to use, from existing libraries, data structures such as vectors, lists, maps, etc. However, you are not allowed to use already-implemented graphs.

You have use one of the following implementations for the graph – assigned to you during the first laboratory:

1. Adjacency matrix
2. List of neighbours
3. Double list of neighbours

Implement the following operations:
For every operation write the time complexity of the operation as a comment.

- ***init*** – creates a new empty graph (no vertices or edges)
- ***add_vertex*** – adds a new vertex to the graph
- ***add_edge*** – adds a new edge to the graph, between 2 existing vertices. If the vertices given do not exist it raises an error. If the edge already exists it also raises an eror (it is a simple graph, not a multigraph). **Note:** this is a directed graph so the edges have an orientation. E.g. an edge from vertwx 1 to vertex 2 is different from an edge from vertex 2 to vertex 1.
- ***remove_edge*** – removes the given edge (given by edge_id, or by pair <initial vertex, terminal vertex>). raises error if the edge is incorrect.
- ***remove_vertex*** – removes a given vertex from the graph, also removin all edges (both inbound and outbound) from the graph. Raises error if the vertex is incorrect.
- ***create_random*** – given a number of vertices *n* and a number of edges *m* it creates a random graph with vertices 0 to *n-1* and *m* edges randomly linking the *n* vertices. If the number of edges is greater that *n(n-1)* it raises an error (the maximum number of edges in a simple directed graph).
- ***get_n*** – returns the number of vertices
- ***get_m*** – returns the number of edges
- ***deg(x)*** – given *x*, a vertex of the graph, return the total degree of the vertex. raises error if x is not in the graph. **Bonus**: add another parameter to control if returned value is the in degree, the out degree or the total degree (+0.5p)
- ***is_edge*** – given 2 vertices check if there is an edge between the 2 vertices. If you have an edge_id, return the edge_id of the edge. Raises error if any of the vertices are not correct.

- ***outbound_edges*** – given a vertex of the graph return a list or a similar iterable that contain all the edges that are outbound of the vertex – it returns the edge_ids if you use them otherwise it returns just the endpoint vertex of the edge. **IMPORTANT:** if the returned lis tis modified in any way the graph should NOT be modified (i.e. make sure that the graph cannot be changed by mistake with this function). Raise an error if the vertex is not in the graph.
- **Bonus: *inbound_edges*** – same as outbound but returns the inbound edges (0.5p)
- if you use edge ids: ***egde_endpoints*** – returns the endpoints of the given edge (initial and terminal vertices of the edge). Raises error if the edge i dis not valid.
- *copy_graph* – creates a copy of the graph, so that if the copy is modified, the initial graph is NOT modified.
- **__str__** – prints the graph in the following way: first on one line prints a list with all the vertices, then prrints all edges (in any order) on a new line, printing adge_id: inital_vertex terminal_vertex. Print the edge_id only f you use it.
  for example you can print something like:
  ```
  0 1 2 3 4 5
  e1: 1 2
  e2: 2 1
  e3: 3 4
  e4: 4 5
  e5: 5 2
  ```