LDD2-Project: Session 2

Aims of this session: To begin manipulating graphs.

When defining methods: Don't forget the *docstring*, which goes under the def line between triple apostrophes. For example:

```
def some_method(self, some_parameters):
        '''
        some_parameters: some_type; some description;
        what the method does …
        '''
        some_commands
```

The previous tutorial should be completed before continuing with this one. We stopped at a function that allowed us to add a node to the graph. Now we're going to see how to remove one.

Since the getters and setters are in place, we are going, in general, to avoid accessing the attributes of an instance of a class directly. If we want to know the id of a node n, for example, we'll use n.get_id() rather than n.id. It's less concise, but more robust.

Exercise 1:
In node, construct the following methods:

1. remove_parent_once and remove_child_once
2. remove_parent_id and remove_child_id

The first two remove 1 occurrence (1 multiplicity) of the id, given as a parameter. In the last two, *all* occurrences of the id are removed. In both cases, if the multiplicity falls to 0, the key is removed from the dictionary.

Exercise 2:
In open_digraph, implement the following methods:

1. remove_edge(self, src, tgt) (src and tgt are ids)
2. remove_parallel_edges (self, src, tgt) (src and tgt are ids)
3. remove_node_by_id

You can also implement the __delitem__ method for one or more of these methods.Hint: x = d.pop(k) is used to remove the key k from the dictionary d and store the associated value in x.

The first function should remove 1 edge only; the second all the edges from src to tgt. The last function should remove the edges associated with the specified node, as well as the node itself.

Implement the methods remove_edges, remove_several_parallel_edges and remove_nodes_by_id, which generalise the previous methods and address several edges/nodes directly. One can equally do 2-in-1 using *args.

As an argument to remove_edges and remove_several_ parallel_edges, we expect something like a list of (src,tgt) pairs.

Exercise 3:
To make sure you don't get just anything after manipulating a graph, it can be useful to implement an is_ well_formed method, which verifies if a graph is well defined. For a graph to be well formed, it should satisfy the following properties:

1. each input and output node must be in the graph (i.e. its id should be a key in nodes)
2. each input node must have a single child (of multiplicity 1) and no parent
3. each output node must have a single parent (of multiplicity 1) and no children
4. each key in nodes corresponds to a node which has the key as id
5. if j has i as a child, with multiplicity m, then i must have j as a parent, with multiplicity m, and vice-versa.

Implement the method assert_is_well_formed, which returns an error if the graph is not well formed.

Exercise 4:
Define the add_input_node method, which creates a new node, to be defined as an input, and which points to the node whose id is specified as a parameter. (The label of the new node can be left empty). Define the add_output_node method in a similar way.

Note: there is a condition to ensure that the graph remains well-formed. Find it and implement it (return an error if the condition is not met).

Exercise 5:
Perform a few tests to check that :

1. is_well_formed accepts good graphs and rejects poorly formed ones
2. Adding or removing a node leaves a well-formed graph
3. Adding or removing an edge leaves a well-formed graph (without taking input/output nodes into account)
4. Adding an input/output using the methods in the preceding exercise leaves a well-formed graph