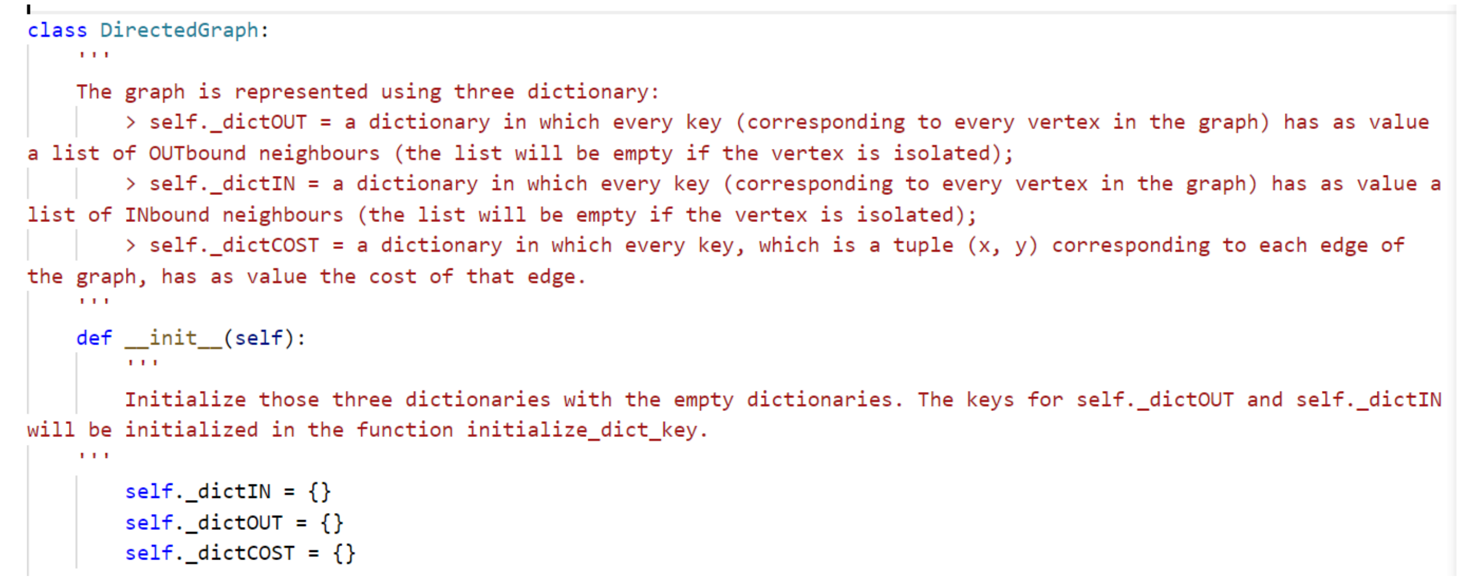
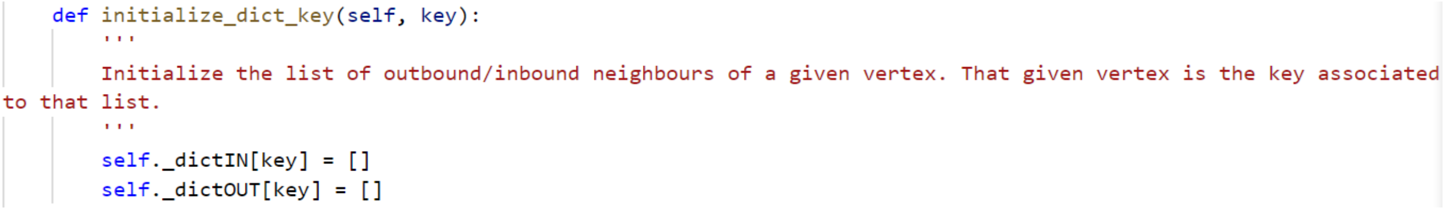
Documentation for Practical Work Number 1

As an internal representation of the graph, I used the ‘collection of neighbours’ implementation (both inbound and outbound), as well as a collection of costs.



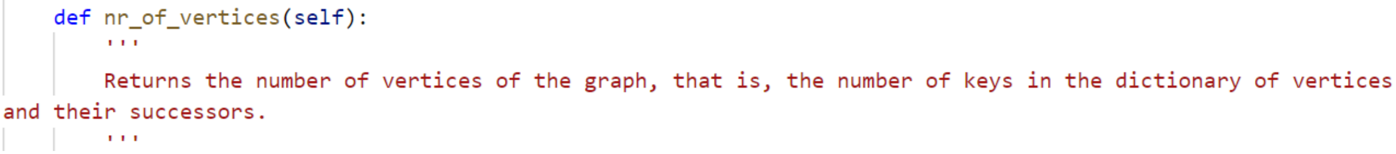
```
class DirectedGraph:
    '''
    The graph is represented using three dictionary:
        > self._dictOUT = a dictionary in which every key (corresponding to every vertex in the graph) has as value
a list of OUTbound neighbours (the list will be empty if the vertex is isolated);
        > self._dictIN = a dictionary in which every key (corresponding to every vertex in the graph) has as value a
list of INbound neighbours (the list will be empty if the vertex is isolated);
        > self._dictCOST = a dictionary in which every key, which is a tuple (x, y) corresponding to each edge of
the graph, has as value the cost of that edge.
    '''
    def __init__(self):
        '''
        Initialize those three dictionaries with the empty dictionaries. The keys for self._dictOUT and self._dictIN
will be initialized in the function initialize_dict_key.
    '''
        self._dictIN = {}
        self._dictOUT = {}
        self._dictCOST = {}
```



```
    def initialize_dict_key(self, key):
        '''
        Initialize the list of outbound/inbound neighbours of a given vertex. That given vertex is the key associated
to that list.
        '''
        self._dictIN[key] = []
        self._dictOUT[key] = []
```

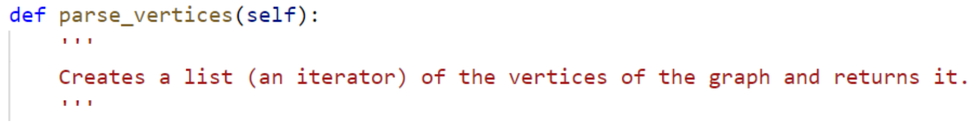
I implemented the required operations as follows:

- Get the number of vertices;



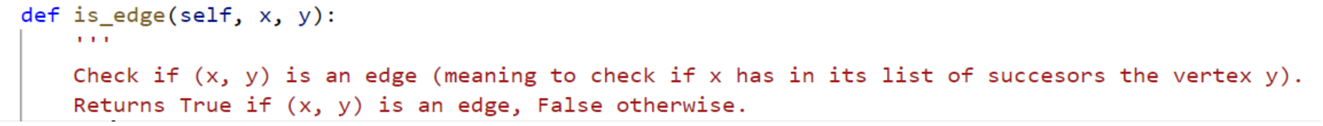
```
    def nr_of_vertices(self):
        '''
        Returns the number of vertices of the graph, that is, the number of keys in the dictionary of vertices
and their successors.
        '''
```

- Parse the set of vertices:



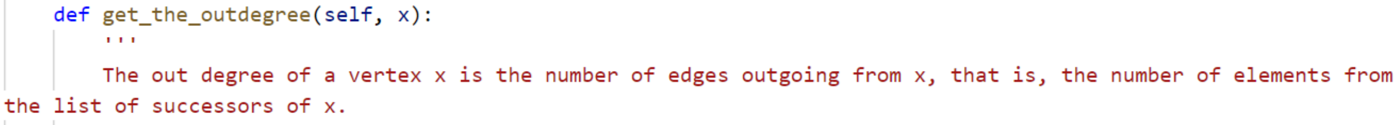
```
def parse_vertices(self):
    '''
    Creates a list (an iterator) of the vertices of the graph and returns it.
    '''
```

- Given two vertices, find out whether there is an edge from the first one to the second one:

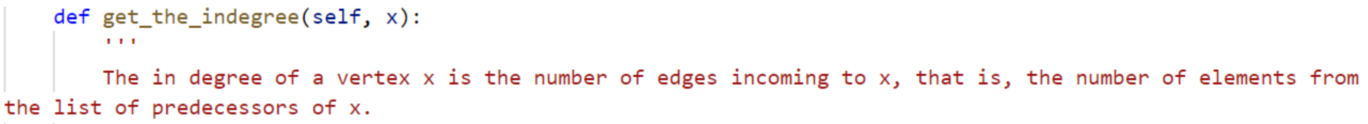


```
def is_edge(self, x, y):
    '''
    Check if (x, y) is an edge (meaning to check if x has in its list of succesors the vertex y).
    Returns True if (x, y) is an edge, False otherwise.
```

- Get the in degree and the out degree of a given vertex:

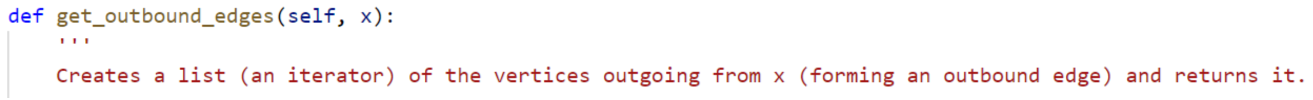


```
    def get_the_outdegree(self, x):
        '''
        The out degree of a vertex x is the number of edges outgoing from x, that is, the number of elements from
the list of successors of x.
```

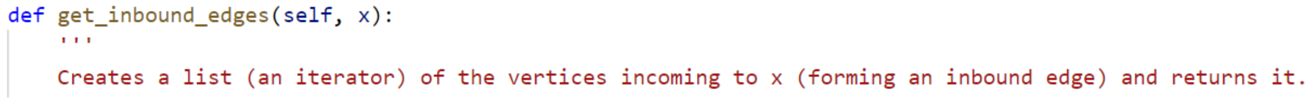


```
    def get_the_indegree(self, x):
        '''
        The in degree of a vertex x is the number of edges incoming to x, that is, the number of elements from
the list of predecessors of x.
```

- Iterate the set of outbound and inbound edges of a specified vertex:

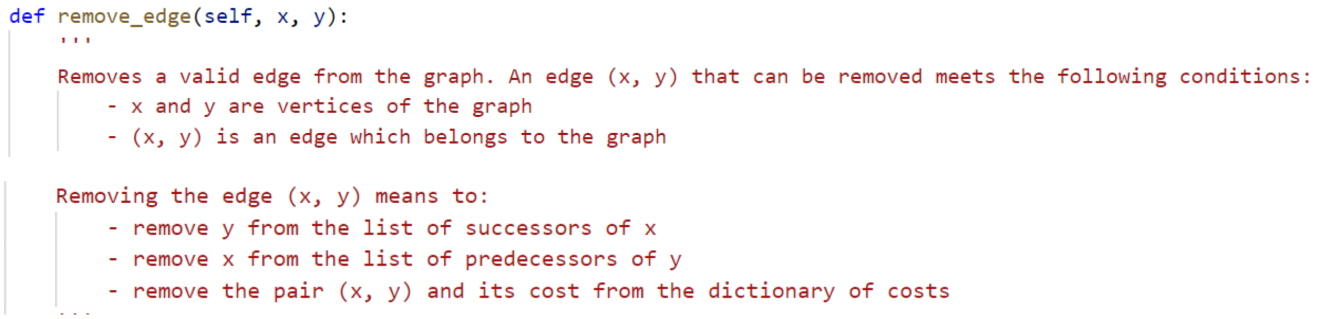


```
def get_outbound_edges(self, x):
    '''
    Creates a list (an iterator) of the vertices outgoing from x (forming an outbound edge) and returns it.
```



```
def get_inbound_edges(self, x):
    '''
    Creates a list (an iterator) of the vertices incoming to x (forming an inbound edge) and returns it.
```

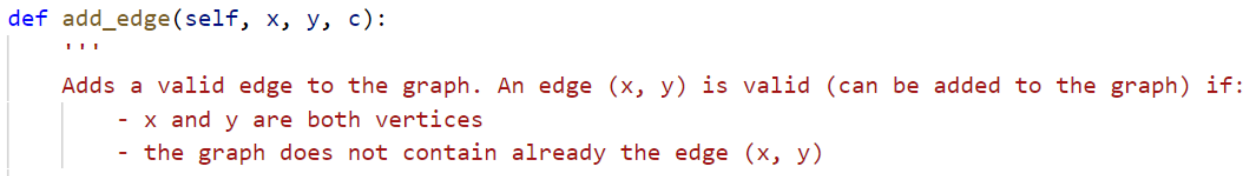
- Remove an edge:



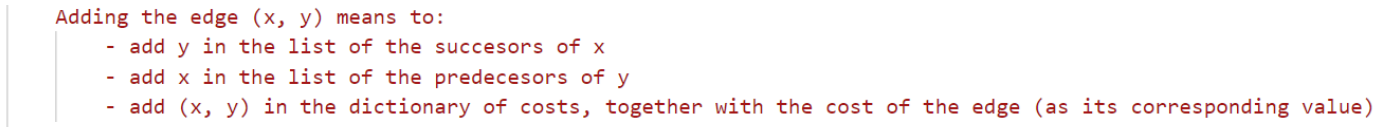
```
def remove_edge(self, x, y):
    '''
    Removes a valid edge from the graph. An edge (x, y) that can be removed meets the following conditions:
        - x and y are vertices of the graph
        - (x, y) is an edge which belongs to the graph

    Removing the edge (x, y) means to:
        - remove y from the list of successors of x
        - remove x from the list of predecessors of y
        - remove the pair (x, y) and its cost from the dictionary of costs
    '''
```

- Add an edge:

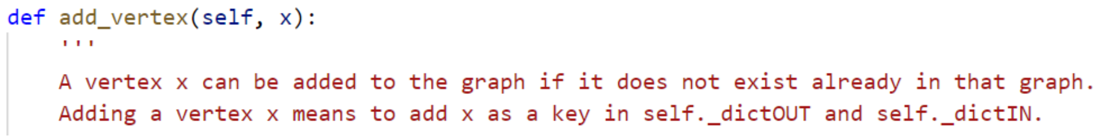


```
def add_edge(self, x, y, c):
    '''
    Adds a valid edge to the graph. An edge (x, y) is valid (can be added to the graph) if:
        - x and y are both vertices
        - the graph does not contain already the edge (x, y)
```



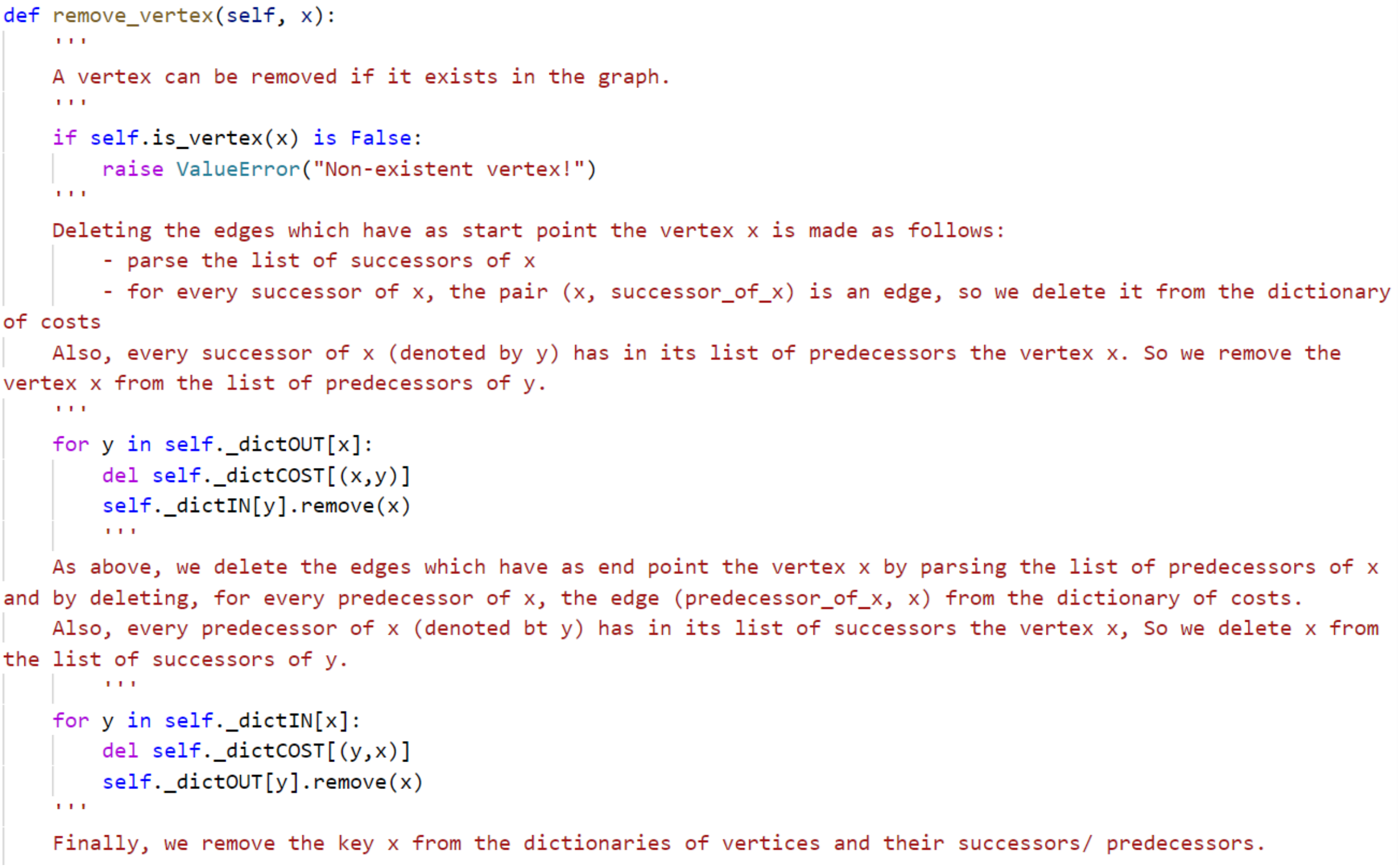
```
    Adding the edge (x, y) means to:
        - add y in the list of the succesors of x
        - add x in the list of the predecesors of y
        - add (x, y) in the dictionary of costs, together with the cost of the edge (as its corresponding value)
```

- Add a vertex:



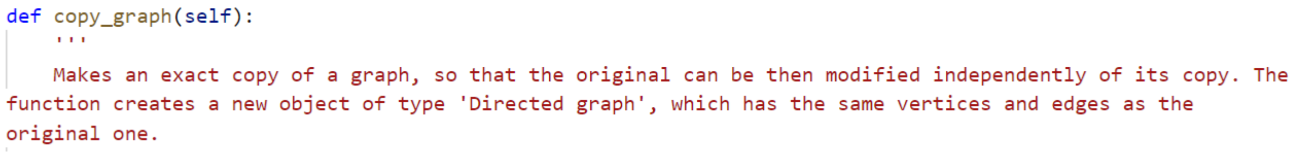
```
def add_vertex(self, x):
    '''
    A vertex x can be added to the graph if it does not exist already in that graph.
    Adding a vertex x means to add x as a key in self._dictOUT and self._dictIN.
```

- Remove a vertex:



```
def remove_vertex(self, x):
    '''
    A vertex can be removed if it exists in the graph.
    '''
    if self.is_vertex(x) is False:
        raise ValueError("Non-existent vertex!")
    '''
    Deleting the edges which have as start point the vertex x is made as follows:
        - parse the list of successors of x
        - for every successor of x, the pair (x, successor_of_x) is an edge, so we delete it from the dictionary
of costs
    Also, every successor of x (denoted by y) has in its list of predecessors the vertex x. So we remove the
vertex x from the list of predecessors of y.
    '''
    for y in self._dictOUT[x]:
        del self._dictCOST[(x,y)]
        self._dictIN[y].remove(x)
        '''
    As above, we delete the edges which have as end point the vertex x by parsing the list of predecessors of x
and by deleting, for every predecessor of x, the edge (predecessor_of_x, x) from the dictionary of costs.
    Also, every predecessor of x (denoted bt y) has in its list of successors the vertex x, So we delete x from
the list of successors of y.
        '''
    for y in self._dictIN[x]:
        del self._dictCOST[(y,x)]
        self._dictOUT[y].remove(x)
    '''
    Finally, we remove the key x from the dictionaries of vertices and their successors/ predecessors.
```

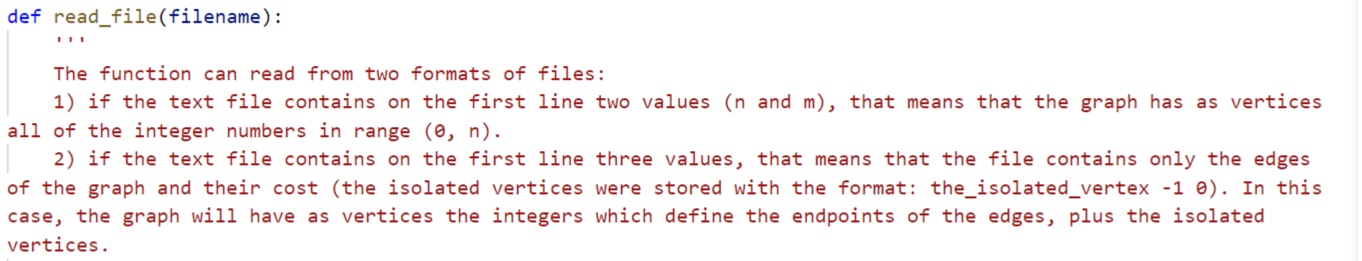
- Copy the graph:



```
def copy_graph(self):
    '''
    Makes an exact copy of a graph, so that the original can be then modified independently of its copy. The
function creates a new object of type 'Directed graph', which has the same vertices and edges as the
original one.
```

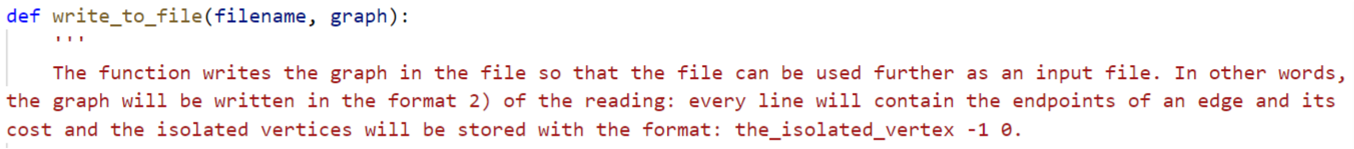
The external function which were implemented are the following:

- Read the graph from a text file:



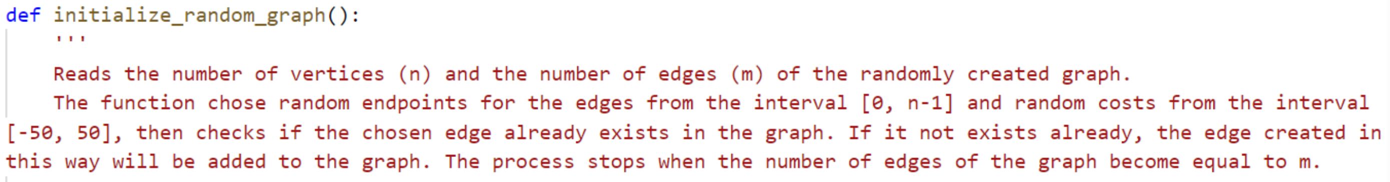
```
def read_file(filename):
    '''
    The function can read from two formats of files:
    1) if the text file contains on the first line two values (n and m), that means that the graph has as vertices
all of the integer numbers in range (0, n).
    2) if the text file contains on the first line three values, that means that the file contains only the edges
of the graph and their cost (the isolated vertices were stored with the format: the_isolated_vertex -1 0). In this
case, the graph will have as vertices the integers which define the endpoints of the edges, plus the isolated
vertices.
```

- Write the graph in a text file:



```
def write_to_file(filename, graph):
    '''
    The function writes the graph in the file so that the file can be used further as an input file. In other words,
the graph will be written in the format 2) of the reading: every line will contain the endpoints of an edge and its
cost and the isolated vertices will be stored with the format: the_isolated_vertex -1 0.
```

- Create a random graph with specified number of vertices and of edges:



```
def initialize_random_graph():
    '''
    Reads the number of vertices (n) and the number of edges (m) of the randomly created graph.
    The function chose random endpoints for the edges from the interval [0, n-1] and random costs from the interval
[-50, 50], then checks if the chosen edge already exists in the graph. If it not exists already, the edge created in
this way will be added to the graph. The process stops when the number of edges of the graph become equal to m.
```