**Practical work no. 1**

**Specification**

We shall define a class named Graph representing a directed graph.

The class Graph will provide the following methods:

**def \_\_init\_\_(self, n):**  
 *"""Creates a graph with n vertices (numbered from 0 to n-1)  
 and no edges"""*

*Postcondition: creates instance of class Graph.*

*Complexity: O(1)*

**def get\_number\_vertices(self):**  
 *"""Returns the number of vertices"""*

*Postcondition: returns an integer representing the number of vertices.*

*Complexity: O(1)*

**def parse\_vertices(self):**  
 *"""Returns an iterable containing all the vertices"""*

*Postcondition: returns an iterator*

*Complexity: Theta(1)*

**def is\_edge(self, x, y):**  
 *"""Check if an edge exists”””  
Precondition: x and y must be valid vertices, else raise exception*

*Postcondition: Returns True if there is an edge from x to y, False otherwise*

*Complexity: Theta(1)*

**def out\_degree(self, x):**  
 *"""Returns the out degree of a specified vertex x”””  
 Precondition: x must be a valid vertex, else raise exception*

*Postcondition: Returns the out degree of the vertex x*

*Complexity: Theta(1)*

**def parse\_vertex\_out(self, x):**  
 *"""Returns an iterable containing the outbound neighbours of x”””  
Precondition: x must be a valid vertex, else raise exception*

*Postcondition: returns an iterator for the outbound vertices of x*

*Complexity: O(1)*

**def in\_degree(self, x):**  
 *"""Returns the in degree of a specified vertex”””  
Precondition: x must be a valid vertex, else raise exception*

*Postcondition: Returns the in degree of the vertex x*

*Complexity: O(1)*

**def parse\_vertex\_in(self, x):** *"""Returns an iterable containing the inbound neighbours of x”””  
Precondition: x must be a valid vertex, else raise exception*

*Postcondition: returns an iterator for the inbound vertices of x*

*Complexity: O(1)*

**def get\_cost(self, x, y):**  
 *"""Get the cost of the edge”””  
Preconditions: x, y must be valid vertices and the edge must exist*

*Postconditions: returns an integer representing the cost of the edge (x,y)*

*Complexity: O(1)*

**def modify\_cost(self, x, y, v):**  
 *"""Modifies the cost of an edge”””  
Preconditions: x, y must be valid vertices and the edge must exist*

*Postconditions: the cost is modified*

*Complexity: Theta(1)*

**def add\_edge(self, x, y, c):**  
 *"""Adds an edge from x to y.”””  
Preconditions: x, y must be valid vertices and the edge must not exist*

*Postconditions: the edge id added*

*Complexity: Theta(1)*

**def add\_edge\_no\_condition(self, x, y, c):** *""”Adds an edge from x to y with the cost c. We assume the input is correct. We use it for reading from a file"""*

*Postconditions: the edge is added*

*Complexity: O(1)*

**def remove\_edge(self, x, y):** *"""Removes an edge from x to y.”””  
Preconditions: x, y must be valid vertices and the edge must exist*

*Postconditions: the edge doesn’t exist anymore*

*Complexity: Theta(1)*

**def add\_vertex(self, x):**  
 *"""Adds a new vertex”””  
Preconditions: the vertex must not exist and must be valid(non-negative)*

*Postconditions: the vertex x no exists*

*Complexity: Theta(1)*

**def remove\_vertex(self, x):**  
 *"""Remove the vertex x.”””  
Precondition: the vertex must exist*

*Postcondition: the vertex is added*

*Complexity: Theta(1)*

**def copy\_graph(self):**  
 *"""Makes a copy of the graph"""*

*Precondition: the graph must exist*

*Postcondition: the graph now exists*

*Compelxity: O(1)*

**def print\_dict(self):**  
 *""" Additional function, role: to see if other functions are working. Prints all 3 dictionaries. """*

**def sort\_dictionaries(self):**  
 *"""Sort dictionary when reading from a modified file. Esthetic role"""*

## Implementation

I use 3 different dictionaries in the class Graph:

self.\_dictIn – representing a dictionary, the keys are the vertices and the values are the vertices which have an inbound edge with the key( value -> key)

self.\_dictOut - representing a dictionary, the keys are the vertices and the values are the vertices which have an outbound edge with the key (key -> value)

self.\_dictCost – representing a dictionary, the keys are a touple representing an existing edge of the graph, and the values are the cost of the edge.