**Practical work no. 1**

**Specification**

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

The class Graph will provide the following methods:

**Graph();**

*"""The implicit constructor of the graph"""*

*Postcondition: creates instance of class Graph.*

*Complexity: O(1)*

**explicit Graph();**

*"""The explicit constructor of the graph. Creates a graph with n vertices (numbered from 0 to n-1) and no edges """*

*Postcondition: creates instance of class Graph.*

*Complexity: O(1)*

**Graph(const Graph&);**

*"""Makes a copy of the graph"""*

*Precondition: the graph must exist*

*Postcondition: the graph now exists*

*Compelxity: O(1)*

**int getNumberOfVertices() const:**  
 *"""Returns the number of vertices"""*

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

*Complexity: O(1)*

**int getCost(int, int):**  
 *"""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)*

**void setCost(int, int, int):**  
 *"""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)*

**std::pair<std::map<int, std::vector<int>>::iterator, std::map<int, std::vector<int>>::iterator> parseVertices():**  
 *"""Returns an iterable containing all the vertices"""*

*Postcondition: returns an iterator*

*Complexity: Theta(1)*

**bool isEdge(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)*

**int outDegree(int):**  
 *"""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)*

**std::pair<std::vector<int>::iterator, std::vector<int>::iterator> parseVertexOut(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)*

**int inDegree(int):**  
 *"""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)*

**std::pair<std::vector<int>::iterator, std::vector<int>::iterator> parse\_vertex\_in(int):** *"""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)*

**void addEdge(int, int, int):**  
 *"""Adds an edge from x to y with the cost c.”””  
Preconditions: x, y must be valid vertices and the edge must not exist*

*Postconditions: the edge id added*

*Complexity: Theta(1)*

**void addEdgeNoCondition(int, int, int):** *""”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)*

**void RemoveEdge(int, int):** *"""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)*

**void AddVertex(int):**  
 *"""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)*

**void removeVertex(int):**  
 *"""Remove the vertex x.”””  
Precondition: the vertex must exist*

*Postcondition: the vertex is added*

*Complexity: Theta(1)*

**~Graph();**

*"""The destructor of the graph"""*

*Postcondition: deletes instance of class Graph.*

*Complexity: O(1)*

## Implementation

I use 3 different “python-dictionaries” in the class Graph. I tried to reproduce the effect of the dictionaries from the python using maps:

std::map<int, std::vector<int>> \_dictIn – a map where the keys are the vertices and the values are a vector of the vertices which have an inbound edge with the key( value -> key)

std::map<int, std::vector<int>> \_dictOut - a map where the keys are the vertices and the values are a vector of the vertices which have an outbound edge with the key (key -> value)

std::map<std::pair<int, int>, int> \_dictCost – a map where the keys are a pair representing an existing edge of the graph, and the values are the cost of the edge.