**Practical work 1 - Documentation**

**Specification:**

We shall define a class called Graph which represents a directed graph.

The class ***Graph*** provides the following methods:

def \_\_init\_\_(self, nr\_vertices=0, nr\_edges=0) -> initializes a graph, implicitly with 0 vertices and 0 edges.

def add\_vertex(self, vertex) -> adds a vertex to the graph.

*The function adds a vertex to the graph  
it adds it to the set of vertices (.\_\_vertices)  
it adds an empty set to the dictionary for outbound  
 neighbours with the key being the vertex  
it adds an empty set to the dictionary for inbound  
 neighbours with the key being the vertex  
it raises an error if the vertex already exists in the graph*

def is\_vertex(self, vertex)

*The function checks whether a vertex is in the graph  
it searches for the vertex in the set of vertices.*

def add\_edge(self, vertex1, vertex2, cost)

T*he function adds an edge and its cost to the graph.  
To the set of outbound neighbours of vertex1, vertex2 is added  
To the set of inbound neighbours of vertex2, vertex1 is added  
To the dictionary that maps the cost of each edge, a key-value pair is added.  
 the key is a tuple of the vertices (vertex1, vertex2)  
 the value is the cost  
The function raises an error if the edge already exists in the graph  
 or if the vertices do not exist in the graph.*

def is\_edge(self, vertex1, vertex2)

*The function checks whether the edge with the vertices vertex1 and vertex2 exists  
the function checks in the outbound neighbours of vertex1 for vertex2 and the  
 inbound neighbours of vertex2 for vertex1*

def get\_number\_of\_vertices(self)

*The function returns the number of vertices in the graph.*

def get\_out\_degree(self, vertex)

*The function returns the out degree of a vertex  
the function raises an error if the vertex doesn't exist in the graph*

def get\_in\_degree(self, vertex)

*The function returns the in degree of a vertex  
the function raises an error if the vertex doesn't exist in the graph*

def get\_edge\_cost(self, vertex1, vertex2)

*The function returns the cost of a given edge  
the cost is taken from the .\_\_cost dictionary  
an error is raised if the edge doesn't exist.*

def set\_edge\_cost(self, vertex1, vertex2, new\_cost)

*The function sets the cost of an edge with the value specified.  
an error is raised if the edge doesn't exist.*

def get\_number\_of\_edges(self)

*The function returns the number of edges in the graph  
it uses the cost dictionary since it holds all of the edges*

def parse\_vertices(self)

*The function returns an iterator to the set of vertices.*

def parse\_inbound\_neighbours(self, vertex)

The function returns an iterator to the set of inbound

neighbours of the vertex.

An error is raised if the vertex doesn't exist in the graph.

def parse\_outbound\_neighbours(self, vertex)

*The function returns an iterator to the set of outbound  
neighbours of the vertex.  
An error is raised if the vertex doesn't exist in the graph.*

def parse\_edges(self)

*The function returns an iterator to the set of edges  
with the cost.*

def remove\_edge(self, vertex1, vertex2)

*The function removes an edge from the graph  
it deletes the entry in the cost dictionary  
it removes vertex2 from vertex1's outbound neighbours  
it removes vertex1 from vertex2's inbound neighbours  
An error is raised if the edge didn't exist in the graph.*

def remove\_vertex(self, vertex)

*The function removes a vertex from the graph.  
It removes all the edges for which the vertex is either an  
inbound node or an outbound one.  
then deletes the key-value item from the dictionaries (.\_\_inbound, .\_\_outbound)  
 with the key= vertex  
lastly the vertex is removed from the list of vertices.  
An error is raised if the vertex didn't exist in the graph.*

def copy(self)

The function returns a deepcopy of the graph.

**Implementation:**

The implementation uses a set for the vertices, dictionaries for the inbound and outbound neighbours of each vertex, a dictionary for the list of edges/cost.

A directed graph, represented as two maps,

using inbound neighbours and outbound neighbours.

In order to implement the cost function, a map for the costs

has been added to the graph.

For ease of use, a set of all the vertices has been added,

making the implementation of certain functionalities simpler.

self.\_\_vertices = set()  
self.\_\_outbound = dict()  
self.\_\_inbound = dict()  
self.\_\_cost = dict()