**Practical work no. 1**

Specification

We shall define a class named DictGraph representing a *directed graph*.

The class Graph will provide the following methods:

*\_\_init\_\_ Graph(n)*

Constructs a graph with *n* vertices and without arcs.

*self*.\_\_in = {}

*self*.\_\_out = {}

*self*.\_\_cost = {}

for i in range(n):

*self*.\_\_in[i]=[]

*self*.\_\_out[i] = []

*getNumberOfVertices()*

Returns the number of vertices

return len(*self*.\_\_in.keys())

*addVertex()*

Inserts a new vertex into the graph. Precondition: the vertex v is not already in the graph

if v in *self*.\_\_in or v in *self*.\_\_out:

return False

*self*.\_\_in[v] = []

*self*.\_\_out[v] = []

return True

*removeVertex(v)*

Removes the given vertex v. Precondition: v must be a vertex of the graph

for k in *self*.\_\_out.keys():

if v in *self*.\_\_in[k]:

*self*.removeEdge(v,k)

*self*.\_\_in.pop(v)

*self*.\_\_out.pop(v)

*addEdge(x,y)*

Inserts an edge from *x* towards *y*. Precondition: there must be no edge from *x to y.*

if *self*.isEdge(x, y):

return False

*self*.\_\_out[x].append(y)

*self*.\_\_in[y].append(x)

*self*.\_\_cost[(x,y)] = c

return True

*removeEdge(x,y)*

Removes the edge going from x to y. Precondition: (x,y) is an edge of the graph

if not *self*.isEdge(x, y):

return False

*self*.\_\_out[x].remove(y)

*self*.\_\_in[y].remove(x)

del *self*.\_\_cost[(x,y)]

return True

*isEdge(x,y)*

Checks whether there is an edge going from x to y. Precondition: x,y valid vertices

*return y in self.\_\_out[x] or x in self.\_\_in[y]*

*getInDegree(v)*

Returns the in degree of vertex v. Precondition: v must be a valid vertex in the graph

*return len(self.\_\_in[v])*

*getOutDegree(v)*

Returns the in degree of vertex v. Precondition: v must be a valid vertex in the graph

return *len(self.\_\_out[v])*

*parseOut(v)*

Returns an iterator over the outbound neighbors of v. Precondition: v must be a valid vertex in the graph.

return *self*.\_\_out[v]

*parseIn(v)*

Returns an iterator over the inbound neighbors of v. Precondition: v must be a valid vertex in the graph.

return *self*.\_\_in[v]

*getCost(x,y)*

Returns the cost of the edge (x,y). Precondition: (x,y) is an edge of the graph.

return *self*.\_\_cost[(x,y)]

*setCost(x,y,c)*

Modifies the cost of the edge (x,y) to be c. Precondition: (x,y) is an edge of the graph.

*self*.\_\_cost[(x,y)] = c

*getNumberOfVertices()*

Returns the number of vertices in the graph.

return len(*self*.\_\_in.keys())

*isValidVertex(v)*

Checks whether the vertex v is valid.

return v in *self*.\_\_in.keys()

Implementation

Each edge (x,y) is represented in the two dictionaries of inbound and outbound neighbours: the first vertex, x, is part of the second vertex’s list of inbound neighbors, and the second vertex,y, is in x’s list of outbound neighbors. The cost of each edge is represented in a third dictionary, that has pairs of vertices – representing edges - as keys, and the corresponding costs as values.

Class Graph will have the following data members:

*in - dictionary*

contains the lists of inbound neighbours of each vertex

*out - dictionary*

contains the lists of outbound neighbours of each vertex

*cost - dictionary*

maps each edge to its cost

Helper functions

*readVertex()*

reads a valid vertex from the keyboard

*readGraph()*

reads the graph from a standard input file