**Representation**

def \_\_init\_\_(self):  
 self.\_dictIn = {} – dictionary of dictionaries – map used to store the in vertices   
 self.\_dictOut = {} - – dictionary of dictionaries – map used to store the out vertices  
 self.\_dictCosts = {} – dictionary with pairs as keys – maps pairs to costs  
 self.\_vertices = 0  
 self.\_edges = 0

**Specification**

Class DoubleDictGraph provides the following methods:

def \_\_init\_\_(self)

Constructs a graph without vertices or arcs.

def vertices(self)

Returns the number of vertices.

def edges(self)

Returns the number of edges.

def is\_edge(self,x, y)

Checks whether or not there is an arc between x and y.

def is\_vertice(self,n)

Checks whether or not n is a vertex.

def add\_vertex(self)

Adds a new vertex to the graph.

def remove\_vertex(self, vertex)

Removes the vertex n from the graph.

Precondition: n is a vertex .

def add\_edge(self, x, y, cost)

Adds an edge to the graph.

Precondition: x and y are existent vertices and the edge x-y doesn’t exist.

def remove\_edge(self, x, y)

Removes an edge from the graph.

Precondition: x-y is an edge.

def get\_vertices(self)

Returns a list of all vertices.

def get\_in\_degree(self, vertex)

Returns the in degree of a given vertex.

Precondition: vertex is in the graph.

def get\_out\_degree(self, vertex)

Returns the out degree of a given vertex.

Precondition: vertex is in the graph.

def parse\_outbound(self, vertex)

Returns the list of outbound neighbors of a given vertex.

Precondition: vertex is in the graph.

def parse\_inbound(self, vertex)

Returns the list of inbound neighbors of a given vertex.

Precondition: vertex is in the graph.

def get\_cost(self, x, y)

Returns the cost of a given edge.

Precondition: x and y are vertices in the graph.

def modify\_cost(self,x, y,newValue)

Changes the cost of a given edge.

Precondition: the edge is in the graph.

def copy(self)

Returns a deep copy of the graph.

def get\_costs(self):

Returns the dictionary of costs.

External functions:

def loadGraphs(graph, filename)

Loads a graph from a text file in the memory.

def storeGraph(graph, filename)

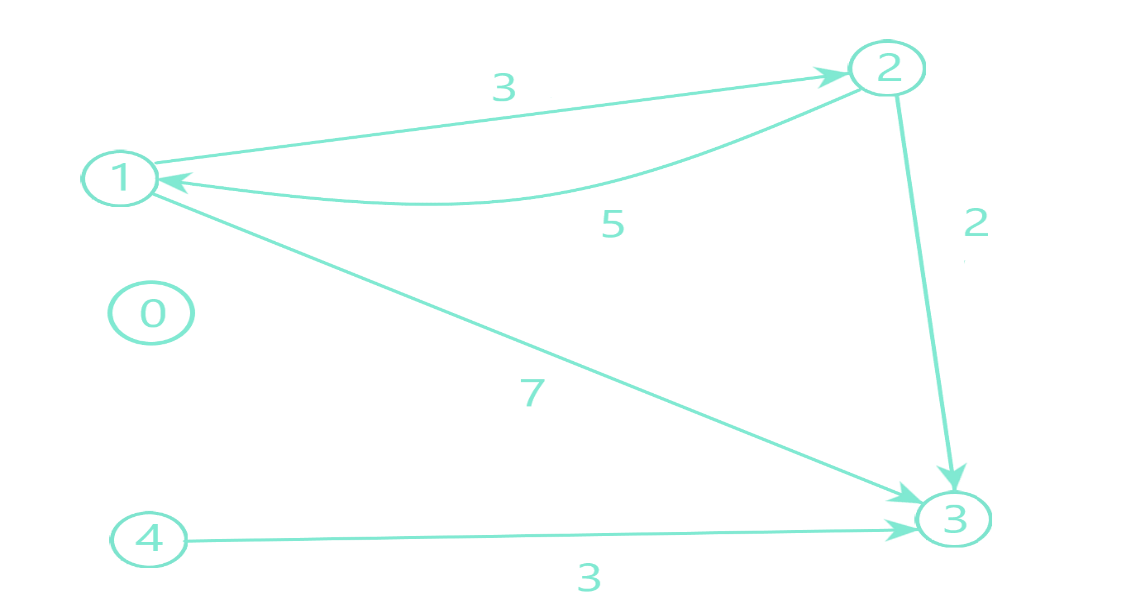
Stores a graph from memory to a text file

def generateRandomGraph(vertices, edges)

Returns a random generated graph with a given number of vertices and edges

**Examples**

The following graph can be obtained by applying the add\_vertex() method 5 times and the add\_edge method in the following way: add\_edge(1,2,3) , add\_edge(1,3,7) , add\_edge(2,1,5), add\_edge(2,3,2), add\_edge(4,3,3).



For this graph the initial values will be:

self.\_dictIn = {

self.\_vertices = 5  
self.\_edges = 5

self.\_dictCosts = {

(1, 2) : 3,

(1, 3) : 7,

(2, 1) : 5,

(2, 3) : 2,

(4, 3) : 3

}

self.\_dictIn = {

0 : {}

1 : {0 : 2}

2 : { 0 : 1}

3 : {0 : 1, 1 : 2, 2:4}

4: {}

}

self.\_dictOut = {

0 : {}

1 : {0 : 2, 1 : 3}

2 : {0 : 1, 1 : 3}

3: {}

4 : {0 : 3}

}

0 : {}

1 : {0 : 2}

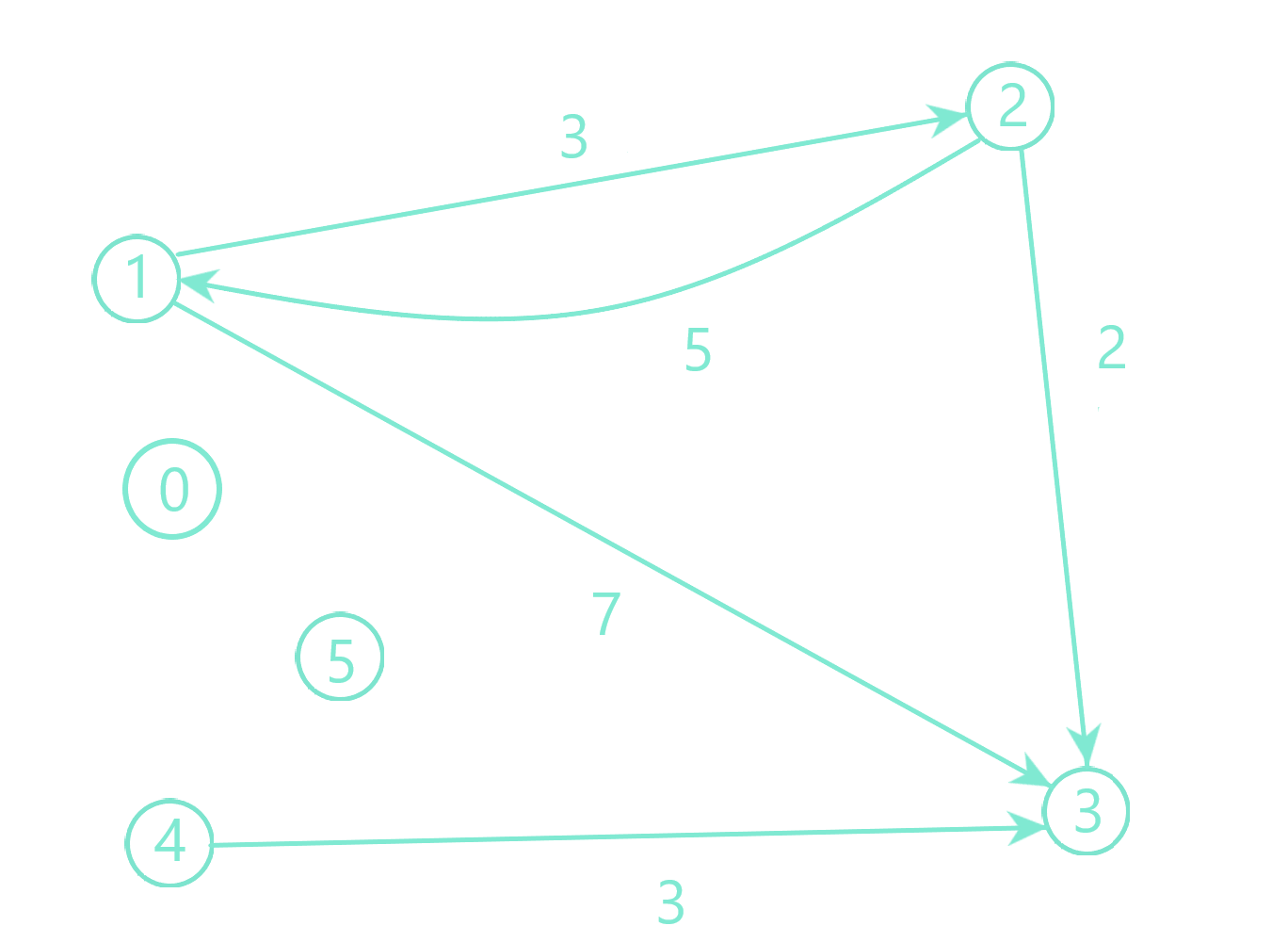
2 : { 0 : 1}

3 : {0 : 1, 1 : 2, 2:4}

4: {}

}

For the same graph if I call add\_vertex method:



self.\_dictOut = {

0 : {}

1 : {0 : 2, 1 : 3}

2 : {0 : 1, 1 : 3}

3: {}

4 : {0 : 3},

5 : {}

}

self.\_dictIn = {

0 : {}

1 : {0 : 2}

2 : { 0 : 1}

3 : {0 : 1, 1 : 2, 2:4}

4: {},

5: {}

}

self.\_dictCosts = {

(1, 2) : 3

(1, 3) : 7

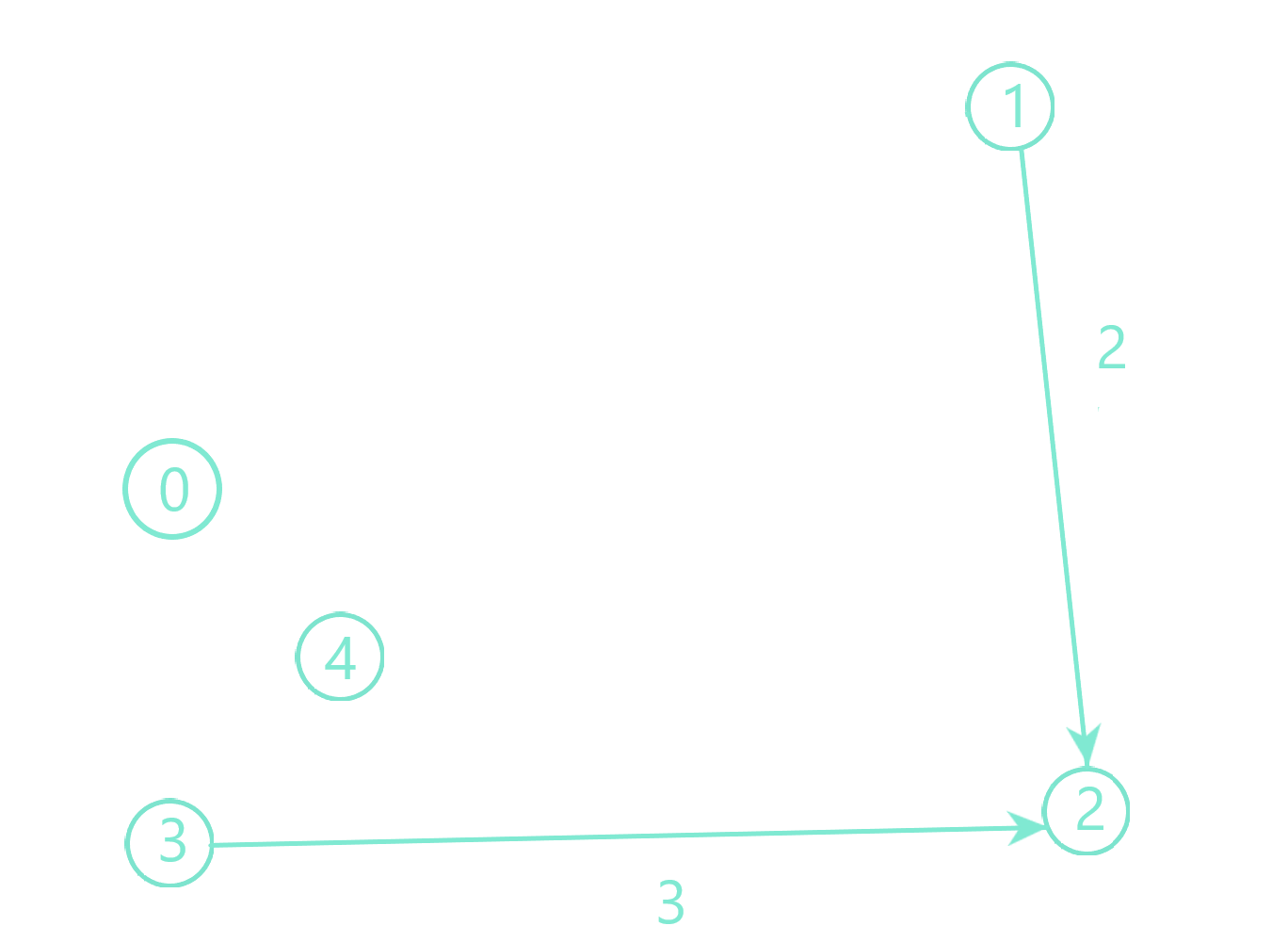
(2, 1) : 5

(2, 3) : 2

(4, 3) : 3

}

self.\_vertices = 6  
self.\_edges = 5

Also if we call remove\_vertex(1) on the previous graph we obtain the following values:

self.\_dictCosts = {

(1, 2) : 2

(3, 2) : 3

}

self.\_dictOut = {

0 : {}

1 : {0 : 2}

2 : {}

3: {0 : 2}

4 : {},

}

self.\_dictIn = {

0 : {}

1 : {}

2 : { 0 : 1, 1 : 3}

3 : {}

4: {},

}

self.\_vertices = 5  
self.\_edges = 2