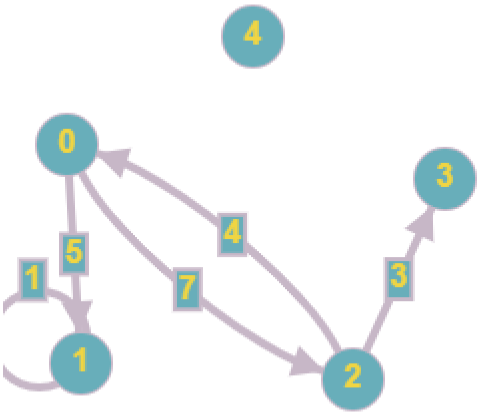
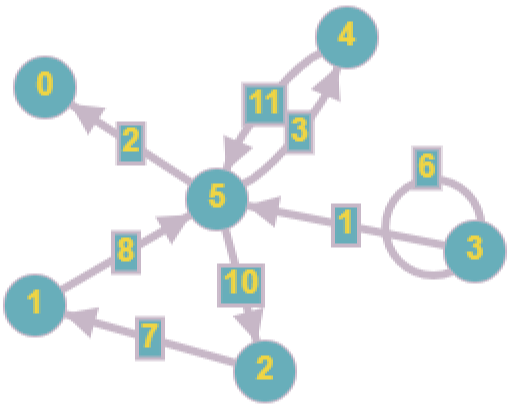
1. Graph representation

For this practical work, I have chosen the directed graph implementation which uses a double list of neighbours for each vertex. In addition, we use a list of edges that registers the cost of the edge, identified by the vertices of the edge. I have chosen this representation, as it has a reduced time complexity for parsing the inbound/outbound neighbours of a given vertex.

Graphic models of this representation:



dIn = {	dOut = {	dCosts = {
0: [2]	0: [1, 2]	(0, 1): 5
1: [0, 1]	1: [1]	(0, 2): 7
2: [0]	2: [0, 3]	(1, 1): 1
3: [2]	3: [ ]	(2, 0): 4
4: [ ]	4: [ ]	(2, 3): 3
}	}	}



dIn = {	dOut = {	dCosts = {
0: [5]	0: [ ]	(1, 5): 8
1: [2]	1: [5]	(2, 1): 7
2: [5]	2: [1]	(3, 3): 6
3: [3]	3: [3, 5]	(3, 5): 1
4: [5]	4: [5]	(4, 5): 11
5: [1, 3, 4]	5: [0, 2, 4]	(5, 0): 2
}	}	(5, 2): 10
		(5, 4): 3
		}

2. Python Implementation

The directed graph is represented by a class named `DoubleListGraph`:

```
class DoubleListGraph():
    def __init__(self,n):
        self._dIn={}
        self._dOut={}
        self._dCosts={}
        for i in range(n):
            self._dIn[i]=[]
            self._dOut[i]=[]
```

where n represents the number of vertices of the graph.

We also need an auxiliary class named `initGraph`, which initialises a graph read from a file:

```
class initGraph():
    def __init__(self,n,m,graph,filename):
        self.graph=graph(n)
        self._filename=filename
        self.__loadFile(filename,m)
```

where n represents the number of vertices, m is the number of edges, graph is the graph to be initialised and filename is the name of the file from which the graph is read, and another auxiliary class named `initRandomGraph`, which initialises a random graph:

```
class initRandomGraph():
    def __init__(self,n,m,graph,filename):
        self.graph=graph(n)
        self.__newGraph(n,m,filename)
```

where n represents the number of vertices, m is the number of edges, graph is the graph to be initialised and filename is the name of the file in which we write the graph obtained.

The class `DoubleListGraph` provides the following methods:

def parseKeys(self):

Returns a list of all the vertices.

def parseKeysCosts(self):

Returns a list of all the edges, described by the start and end vertices.

def parsedCosts(self):

Returns the list of costs, identified by the edge.

def parsedIn(self):

Returns the list of inbound neighbours for each vertex.

def parsedOut(self):

Returns the list of outbound neighbours for each vertex.

def retdIn(self,v):

Returns the list of inbound neighbours for a given vertex v.

def retdOut(self,v):

Returns the list of outbound neighbours for a given vertex v.

def retdCosts(self,x,y):

Returns the cost of an edge described by the start vertex x and the end vertex y.

def isVertex(self,v):

Checks whether a given vertex v is in the graph.

def isEdge(self,x,y):

Checks whether an edge given by the start vertex x and the end vertex y is in the graph.

def addEdge(self,x,y,c):

Adds an edge given by a start vertex x, an end vertex y and a cost c to the graph. If the edge already exists, it will raise an exception.

def modifEdge(self,x,y,c):

Modifies the cost of an edge given by the start vertex x and the end vertex y to the given cost c. If the edge doesn't already exist in order to be modified, it will raise an exception.

def addVertex(self,v):

Adds a vertex v to the graph. If the vertex already exists, it will raise an exception.

def removeEdge(self,x,y):

Removes an edge given by the start vertex x and the end vertex y from the graph. If the edge does not exist in order to be removed, it will raise an exception.

def removeVertex(self,v):

Removes a vertex v from the graph. If the vertex does not exist in order to be removed, it will raise an exception.

def appendToFile(self):

Appends the new graph to a given file after each modification.

def saveFile(self,filename):

Prints the final graph in a given file.

def restoreGraph(self,dIn,dOut,dCosts):

Restores the graph to a previously made copy, replacing each list with the lists corresponding to the graph we want restored.

def numberOfEdges(self):

Returns the number of edges in the graph.

def numberOfVertices(self):

Returns the number of vertices in the graph.

The class `initGraph` provides the following method:

def __loadFile(self,filename,m):

Reads the m edges of the graph from the given file and adds the edges to the graph.

The class `initRandomGraph` provides the following method:

def __newGraph(self,n,m,filename):

Checks if we can create a graph with the given number of vertices and edges. If it is possible, it creates the random graph and writes it to the given file.