Documentación: Lenguaje EZGraph

1. Grafo no dirigido sin pesos

Los grafos no dirigidos sin pesos son modelados como una clase de python, cuyo constructor es el siguiente:

```
class UndirectedGraph:
    def __init__( self , numberOfNodes ):
        if( numberOfNodes <= 0 ):
            print("Error: Number of nodes must be positive")
            exit()
        self.nodes = numberOfNodes
        self.adjacencyList = []
        self.numberOfEdges = 0
        self.adjacencyMatrix = []
        for i in range(numberOfNodes):
            self.adjacencyMatrix.append( [None] * self.nodes )
        for i in range(numberOfNodes):
            self.adjacencyList.append( [] )
```

Dada la creación de la matriz de adyacencia vacía inicial, la complejidad del constructor es O(N^2) siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

Recibe como único parámetro la cantidad de nodos del grafo a crear. Por defecto, sus nodos se numeran de 0 a N-1, siendo N la cantidad de nodos del grafo.

```
NDGraph grafo [8];
```

a. addEdge

Función que permite agregar un arco al grafo.

Implementación en Python

Recibe dos argumentos, que deben ser índices válidos de nodos del grafo. No permite arcos repetidos, ni arcos que empiecen y terminen en el mismo nodo. Retorna un valor booleano que representa si el arco fue creado con éxito.

```
def addEdge( self , node1 , node2 ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        if( node1 == node2 ):
            print("Error: Self loops are not allowed")
            exit()
        if( self.adjacencyMatrix[node1][node2] != None ):
            print("Error: Edge between",node1,"and",node2,"already exists")
            exit()
        if( self.adjacencyMatrix[node2][node1] != None ):
            print("Error: Edge between",node2,"and",node1,"already exists")
            exit()
        self.numberOfEdges = self.numberOfEdges + 1
        self.adjacencyMatrix[node1][node2] = 1
        self.adjacencyMatrix[node2][node1] = 1
        self.adjacencyList[node1].append( node2 )
        self.adjacencyList[node2].append( node1 )
        return True
```

Tiene complejidad O(1).

Utilización en EZGraph

Recibe como parámetros los índices de los nodos que serán conectados por un nuevo arco bidireccional.

```
NDGraph grafo [8];
grafo.addEdge(0,5);
```

b. deleteEdge

Función que permite eliminar un arco del grafo.

Implementación en Python

Recibe dos argumentos, que deben ser índices válidos de nodos del grafo. Retorna un valor booleano que representa si el arco fue eliminado con éxito.

```
    def deleteEdge( self , node1, node2 ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        for edge in self.adjacencyList[node1]:
            if edge == node2:
                self.adjacencyList[node1].remove( edge )
                self.adjacencyMatrix[node1][node2] = None
        for edge in self.adjacencyList[node2]:
            if edge == node1:
                self.adjacencyList[node2].remove( edge )
                self.adjacencyMatrix[node2][node1] = None
                self.numberOfEdges = self.numberOfEdges - 1
                return True
        print("Error: Edge between",node1,"and",node2,"does not exist")
        exit()
```

Tiene complejidad O(M), siendo M la cantidad de arcos del grafo, dado que se debe recorrer la lista de adyacencia para encontrar el arco y después se elimina de una lista estándar de Python.

Utilización en EZGraph

Recibe como parámetros los índices de los nodos del arco bidireccional que será eliminado del grafo. Los índices pueden ser dados en cualquier orden.

```
NDGraph grafo [8];
grafo.addEdge(0,5);
grafo.deleteEdge(0,5);
```

c. getNumEdges

Función que retorna la cantidad de arcos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNumEdges( self ):
        return self.numberOfEdges
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de arcos en el grafo.

```
NDGraph grafo [8];
...
```

```
edges = grafo.getNumEdges();
```

d. getNodes

Función que retorna la cantidad de nodos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNodes( self ):
        return self.nodes
```

Este valor se mantiene constante en el grafo. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de nodos en el grafo.

```
NDGraph grafo [8];
nodes = grafo.getNodes();
```

e. getEdges

Función que retorna una lista con todos los arcos del grafo

Implementación en Python

No recibe ningún argumento. Retorna una lista con todos los arcos actuales del grafo.

```
def getEdges( self ):
```

```
        allEdges = []
        for node1 in range(self.nodes):
            for node2 in self.adjacencyList[node1]:
                if( node1 < node2 ):
                    allEdges.append( [node1,node2] )
        return allEdges
```

Calcula la lista recorriendo la lista de adyacencia del grafo. Al ser no dirigido, solo incluye una vez cada arco al siempre poner primero el nodo de menor índice. Tiene complejidad O(N+M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

No recibe ningún argumento. Retorna una lista de listas de tamaño 2, todos los arcos del grafo.

```
NDGraph grafo [8];
...
edges = grafo.getEdges();
```

f. getMatrix

Función que retorna la matriz de adyacencia del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getAdjacencyMatrix( self ):
        return self.adjacencyMatrix
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1). Sus valores posibles para una posición i,j son None sí no existe arco entre el nodo i y el nodo j y 1 sí existe un arco entre estos nodos.

Utilización en EZGraph

No recibe ningún argumento. Retorna una matriz, la matriz de adyacencia del grafo.

```
NDGraph grafo [8];
nodes = grafo.getMatrix();
```

g. getDistanceFromNode

Función que retorna la distancia desde un nodo específico a todos los nodos.

Implementación en Python

Recibe un argumento, que debe ser un índice válido de un nodo del grafo. Retorna una lista con la distancia desde el nodo inicial dado a todos los nodos del grafo. Sí no es posible llegar algún nodo desde el nodo inicial, retorna infinito.

```
def minDistanceFromSourceToAll( self , source ):
        if( source < 0 or source >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        q = queue.Queue()
        q.put( source )
        while( q.empty() == False ):
            currentNode = q.get()
            for nextNode in self.adjacencyList[currentNode]:
                if( distance[nextNode] > distance[currentNode] + 1 ):
                    distance[nextNode] = distance[currentNode] + 1
                    q.put( nextNode )
        return distance
```

Utiliza una BFS para calcular las distancias, a partir de la estructura de datos Queue, que es una cola FIFO. Su complejidad es de O(N+M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicial. Retorna una lista de N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias del nodo inicial al nodo i-ésimo.

```
NDGraph grafo [8];
...
distances = grafo.getDistanceFromNode(3);
```

h. getAllDistances

Función que retorna una matriz con las distancias entre todos los pares de nodos.

Implementación en Python

No recibe ningún argumento. Retorna una matriz con las distancias desde todos los nodos a todos los nodos del grafo. Sí no es posible llegar algún nodo desde otro, retorna infinito.

```
def minDistanceFromAllToAll( self ):
        distance = [[float('inf') for col in range(self.nodes)] for row in range(self.nodes)]
        for node1 in range( self.nodes ):
            distance[node1][node1] = 0
            for edge in self.adjacencyList[node1]:
                node2 = edge
                distance[node1][node2] = min( distance[node1][node2] , 1 )
        for k in range( self.nodes ):
            for i in range( self.nodes ):
                for j in range( self.nodes ):
                    distance[i][j] = min( distance[i][j] , distance[i][k] + distance[k][j] )
```

```
        for k in range( self.nodes ):
            for i in range( self.nodes ):
                for j in range( self.nodes ):
                    if( distance[i][j] > distance[i][k] + distance[k][j] ):
                        print("Error: Negative Weight Cycle")
                        exit()
        return distance
```

Utiliza el algoritmo de Floyd Warshall, por lo tanto tiene complejidad O(N^3), siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

No recibe argumentos. Retorna una matriz de N x N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias entre cualquier par de nodos. Para los nodos i, j tales que sea imposible ir de ia j, la distancia se muestra como infinito.

```
NDGraph grafo [8];
...
distances = grafo.getAllDistances();
```

i. getDistance

Función que retorna la distancia mínima entre dos nodos dados.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna un entero con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPairDistance( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
```

```
        distance = self.minDistanceFromSourceToAll( source )
        return distance[ destination ]
```

Utiliza como base getDistanceFromNode, por lo que su complejidad es la misma, O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna un entero, la distancia mínima desde el nodo inicial al nodo final. Sí no es posible llegar del nodo inicial al nodo final, retorna infinito.

```
NDGraph grafo [8];
...
distances = grafo.getAllDistances();
```

j. getShortestPath

Función que retorna los nodos de un camino mínimo entre el nodo inicial y el nodo final.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna una lista que representa un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPath( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        q = queue.Queue()
        q.put( source )
        previousNode = [-1] * self.nodes
```

```
        while( q.empty() == False ):
            currentNode = q.get()
            for nextNode in self.adjacencyList[currentNode]:
                if( distance[nextNode] > distance[currentNode] + 1 ):
                    distance[nextNode] = distance[currentNode] + 1
                    previousNode[nextNode] = currentNode
                    q.put( nextNode )
        if( previousNode[destination] == -1 ):
            print( "Error: No path between the nodes")
            exit()
        path = deque([destination])
        currentNode = destination
        while( currentNode != source ):
            currentNode = previousNode[currentNode]
            path.appendleft(currentNode)
        return list(path)
```

Utiliza una BFS que lleva además el registro del nodo inmediatamente anterior para cada nodo. Con esta información, al final se reconstruye el camino mínimo y se retorna la lista. Su complejidad es O(N+M) dado el uso de una BFS con una cola FIFO, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna una lista, un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
NDGraph grafo [8];
...
path = grafo.getShortestPath(6,1);
```

k. BFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda BFS, también conocida como búsqueda a lo ancho.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda BFS.

```
def BFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        q = queue.Queue()
        visited[node] = True
        q.put( node )
        while( q.empty() == False ):
            currNode = q.get()
            order.append( currNode )
            for nextNode in self.adjacencyList[currNode]:
                if visited[nextNode] == False:
                    visited[nextNode] = True
                    q.put( nextNode )
        return order
```

Utiliza una BFS que lleva además el registro de los nodos visitados. Su complejidad es O(N+M) dado el uso de una BFS con una cola FIFO, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda BFS que inicia desde el nodo.

```
NDGraph grafo [8];
...
BFSorder = grafo.BFS(7);
```

I. DFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda DFS , también conocida como búsqueda en profundidad.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda DFS.

```
def DFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        def DFSrecursion( node ):
            nonlocal order
            nonlocal visited
            order.append( node )
            visited[node] = True
            for nextNode in self.adjacencyList[node]:
                if visited[nextNode] == False:
```

```
                    DFSrecursion( nextNode )
        DFSrecursion( node )
        return order
```

Utiliza recursión para visitar los nodos primero en profundidad. Visita cada nodo una vez y cada arista la considera una vez, por lo que su complejidad es de O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda DFS que inicia desde el nodo.

```
NDGraph grafo [8];
...
DFSorder = grafo.DFS(0);
```

2. Grafo no dirigido con pesos

Los grafos no dirigidos con pesos son modelados como una clase de python, cuyo constructor es el siguiente:

```
class UndirectedWeightedGraph:
    def __init__( self , numberOfNodes ):
        if( numberOfNodes <= 0 ):
            print("Error: Number of nodes must be positive")
            exit()
        self.nodes = numberOfNodes
        self.adjacencyList = []
        self.numberOfEdges = 0
        self.adjacencyMatrix = []
        for i in range(numberOfNodes):
            self.adjacencyMatrix.append( [None] * self.nodes )
        for i in range(numberOfNodes):
```

```
            self.adjacencyList.append( [] )
```

Dada la creación de la matriz de adyacencia vacía inicial, la complejidad del constructor es O(N^2) siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

Recibe como único parámetro la cantidad de nodos del grafo a crear. Por defecto, sus nodos se numeran de 0 a N-1, siendo N la cantidad de nodos del grafo.

```
NDWGraph grafo [8];
```

a. addEdge

Función que permite agregar un arco al grafo.

Implementación en Python

Recibe tres argumentos, que deben ser índices válidos de nodos del grafo, y un peso que puede ser entero o decimal. No permite arcos repetidos, ni arcos que empiecen y terminen en el mismo nodo. Retorna un valor booleano que representa si el arco fue creado con éxito.

```
def addEdge( self , node1 , node2 , weight ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        if( node1 == node2 ):
            print("Error: Self loops are not allowed")
            exit()
        if( self.adjacencyMatrix[node1][node2] != None ):
            print("Error: Edge between",node1,"and",node2,"already exists")
            exit()
        if( self.adjacencyMatrix[node2][node1] != None ):
```

```
            print("Error: Edge between",node2,"and",node1,"already exists")
            exit()
        self.numberOfEdges = self.numberOfEdges + 1
        self.adjacencyMatrix[node1][node2] = weight
        self.adjacencyMatrix[node2][node1] = weight
        self.adjacencyList[node1].append( [node2 , weight] )
        self.adjacencyList[node2].append( [node1 , weight] )
        return True
```

Tiene complejidad O(1).

Utilización en EZGraph

Recibe como parámetros los índices de los nodos que serán conectados por un nuevo arco bidireccional, y el peso del arco que los conecta.

```
NDWGraph grafo [8];
grafo.addEdge(0,5,2.5);
```

b. deleteEdge

Función que permite eliminar un arco del grafo.

Implementación en Python

Recibe dos argumentos, que deben ser índices válidos de nodos del grafo. Retorna un valor booleano que representa si el arco fue eliminado con éxito.

```
def deleteEdge( self , node1, node2 ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
```

```
        for edge in self.adjacencyList[node1]:
            if edge[0] == node2:
                self.adjacencyList[node1].remove( edge )
                self.adjacencyMatrix[node1][node2] = None
        for edge in self.adjacencyList[node2]:
            if edge[0] == node1:
                self.adjacencyList[node2].remove( edge )
                self.adjacencyMatrix[node2][node1] = None
                self.numberOfEdges = self.numberOfEdges - 1
                return True
        print("Error: Edge between",node1,"and",node2,"does not exist")
        exit()
```

Tiene complejidad O(M), siendo M la cantidad de arcos del grafo, dado que se debe recorrer la lista de adyacencia para encontrar el arco y después se elimina de una lista estándar de Python.

Utilización en EZGraph

Recibe como parámetros los índices de los nodos del arco bidireccional que será eliminado del grafo. Los índices pueden ser dados en cualquier orden.

```
NDWGraph grafo [8];
grafo.addEdge(0,5);
grafo.deleteEdge(0,5);
```

c. getNumEdges

Función que retorna la cantidad de arcos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNumEdges( self ):
        return self.numberOfEdges
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de arcos en el grafo.

```
NDWGraph grafo [8];
...
edges = grafo.getNumEdges();
```

d. getNodes

Función que retorna la cantidad de nodos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNodes( self ):
        return self.nodes
```

Este valor se mantiene constante en el grafo. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de nodos en el grafo.

```
NDWGraph grafo [8];
nodes = grafo.getNodes();
```

e. getEdges

Función que retorna una lista con todos los arcos del grafo

Implementación en Python

No recibe ningún argumento. Retorna una lista con todos los arcos actuales del grafo.

```
def getEdges( self ):
        allEdges = []
        for node1 in range(self.nodes):
            for edge in self.adjacencyList[node1]:
                node2 = edge[0]
                weight = edge[1]
                if( node1 < node2 ):
                    allEdges.append( [node1,node2,weight] )
        return allEdges
```

Calcula la lista recorriendo la lista de adyacencia del grafo. Al ser no dirigido, solo incluye una vez cada arco al siempre poner primero el nodo de menor índice. Tiene complejidad O(N+M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

No recibe ningún argumento. Retorna una lista de listas de tamaño 3, todos los arcos del grafo, con sus respectivos pesos.

```
```

```
NDWGraph grafo [8];
...
edges = grafo.getEdges();
```

f. getMatrix

Función que retorna la matriz de adyacencia del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getAdjacencyMatrix( self ):
        return self.adjacencyMatrix
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1). Sus valores posibles para una posición i,j son None sí no existe arco entre el nodo i y el nodo j y el peso del arco sí existe un arco entre estos nodos.

Utilización en EZGraph

No recibe ningún argumento. Retorna una matriz, la matriz de adyacencia del grafo.

```
NDWGraph grafo [8];
nodes = grafo.getMatrix();
```

g. getDistanceFromNode

Función que retorna la distancia desde un nodo específico a todos los nodos.

Implementación en Python

Recibe un argumento, que debe ser un índice válido de un nodo del grafo. Retorna una lista con la distancia desde el nodo inicial dado a todos los nodos del grafo. Sí no es posible llegar algún nodo desde el nodo inicial, retorna infinito.

```
def minDistanceFromSourceToAll( self , source ):
        if( source < 0 or source >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        for i in range( self.nodes-1 ):
            for node1 in range( self.nodes ):
                for edge in self.adjacencyList[node1]:
                    node2 = edge[0]
                    weight = edge[1]
                    if( distance[node2] > distance[node1] + weight ):
                        distance[node2] = distance[node1] + weight
        for node1 in range( self.nodes ):
            for edge in self.adjacencyList[node1]:
                node2 = edge[0]
                weight = edge[1]
                if( distance[node2] > distance[node1] + weight ):
                    print("Error: Negative Weight Cycle")
                    exit()
        return distance
```

Utiliza el algoritmo de Bellman Ford para calcular las distancias, utilizando ciclos anidados para relajar cada arco N-1 veces. Su complejidad es de O(N*M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicial. Retorna una lista de N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias del nodo inicial al nodo i-ésimo.

```
NDWGraph grafo [8];
...
distances = grafo.getDistanceFromNode(3);
```

h. getAllDistances

Función que retorna una matriz con las distancias entre todos los pares de nodos.

Implementación en Python

No recibe ningún argumento. Retorna una matriz con las distancias desde todos los nodos a todos los nodos del grafo. Sí no es posible llegar algún nodo desde otro, retorna infinito.

```
def minDistanceFromAllToAll( self ):
        distance = [[float('inf') for col in range(self.nodes)] for row in range(self.nodes)]
        for node1 in range( self.nodes ):
            distance[node1][node1] = 0
            for edge in self.adjacencyList[node1]:
                node2 = edge
                distance[node1][node2] = min( distance[node1][node2] , 1 )
        for k in range( self.nodes ):
            for i in range( self.nodes ):
                for j in range( self.nodes ):
                    distance[i][j] = min( distance[i][j] , distance[i][k] + distance[k][j] )
        for k in range( self.nodes ):
            for i in range( self.nodes ):
                for j in range( self.nodes ):
                    if( distance[i][j] > distance[i][k] + distance[k][j] ):
                        print("Error: Negative Weight Cycle")
                        exit()
        return distance
```

Utiliza el algoritmo de Floyd Warshall, por lo tanto tiene complejidad O(N^3), siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

No recibe argumentos. Retorna una matriz de N x N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias entre cualquier par de nodos. Para los nodos i, j tales que sea imposible ir de ia j, la distancia se muestra como infinito.

```
NDWGraph grafo [8];
...
distances = grafo.getAllDistances();
```

i. getDistance

Función que retorna la distancia mínima entre dos nodos dados.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna un entero con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPairDistance( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = self.minDistanceFromSourceToAll( source )
        return distance[ destination ]
```

Utiliza como base getDistanceFromNode, por lo que su complejidad es la misma, O(N*M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna un entero, la distancia mínima desde el nodo inicial al nodo final. Sí no es posible llegar del nodo inicial al nodo final, retorna infinito.

```
NDWGraph grafo [8];
...
distances = grafo.getAllDistances();
```

j. getShortestPath

Función que retorna los nodos de un camino mínimo entre el nodo inicial y el nodo final.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna una lista que representa un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPath( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        previousNode = [-1] * self.nodes
        for i in range( self.nodes-1 ):
            for node1 in range( self.nodes ):
                for edge in self.adjacencyList[node1]:
                    node2 = edge[0]
                    weight = edge[1]
                    if( distance[node2] > distance[node1] + weight ):
                        distance[node2] = distance[node1] + weight
```

```
                    previousNode[node2] = node1
        for node1 in range( self.nodes ):
            for edge in self.adjacencyList[node1]:
                node2 = edge[0]
                weight = edge[1]
                if( distance[node2] > distance[node1] + weight ):
                    print("Error: Negative Weight Cycle")
                    exit()
        if( previousNode[destination] == -1 ):
            print( "Error: No path between the nodes")
            exit()
        path = deque([destination])
        currentNode = destination
        while( currentNode != source ):
            currentNode = previousNode[currentNode]
            path.appendleft(currentNode)
        return list(path)
```

Utiliza el algoritmo de Bellman Ford, pero además lleva además el registro del nodo inmediatamente anterior para cada nodo. Con esta información, al final se reconstruye el camino mínimo y se retorna la lista. Su complejidad es O(N*M) dado el uso de Bellman Ford, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna una lista, un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
NDWGraph grafo [8];
...
path = grafo.getShortestPath(6,1);
```

k. BFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda BFS, también conocida como búsqueda a lo ancho.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda BFS.

```
def BFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        q = queue.Queue()
        visited[node] = True
        q.put( node )
        while( q.empty() == False ):
            currNode = q.get()
            order.append( currNode )
            for edge in self.adjacencyList[currNode]:
                nextNode = edge[0]
                if visited[nextNode] == False:
                    visited[nextNode] = True
                    q.put( nextNode )
        return order
```

Utiliza una BFS que lleva además el registro de los nodos visitados. Su complejidad es O(N+M) dado el uso de una BFS con una cola FIFO, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda BFS que inicia desde el nodo.

```
NDWGraph grafo [8];
...
BFSorder = grafo.BFS(7);
```

I. DFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda DFS , también conocida como búsqueda en profundidad.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda DFS.

```
def DFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        def DFSrecursion( node ):
            nonlocal order
            nonlocal visited
            order.append( node )
            visited[node] = True
            for edge in self.adjacencyList[node]:
                nextNode = edge[0]
```

```
                if visited[nextNode] == False:
                    DFSrecursion( nextNode )
        DFSrecursion( node )
        return order
```

Utiliza recursión para visitar los nodos primero en profundidad. Visita cada nodo una vez y cada arista la considera una vez, por lo que su complejidad es de O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda DFS que inicia desde el nodo.

```
NDWGraph grafo [8];
...
DFSorder = grafo.DFS(0);
```

m. getMinimumSpanningTree

Función que calcula un árbol de mínima expansión de un grafo conexo, es decir, el subconjunto de aristas del grafo tal que formen un árbol con todos los nodos del grafo y la suma de sus pesos sea mínima.

Implementación en Python

No recibe argumentos. Retorna un grafo no dirigido con pesos, un árbol de mínima expansión del grafo actual.

```
def MinimumSpanningTree( self ):
        tree = UndirectedWeightedGraph( self.nodes )
        pq = PriorityQueue()
        visited = [ False ] * self.nodes
        visited[0] = True
        for edge in self.adjacencyList[0]:
            node = edge[0]
```

```
            weight = edge[1]
            if( visited[node] == False ):
                pq.put((weight , (0,node,weight) ))
        while( pq.empty() == False ):
            currentEdge = pq.get()
            node1 = currentEdge[1][0]
            node2 = currentEdge[1][1]
            weight = currentEdge[1][2]
            if ( visited[node2] == True ):
                continue
            visited[node2] = True
            tree.addEdge( node1 , node2 , weight )
            for edge in self.adjacencyList[node2]:
                node3 = edge[0]
                weight = edge[1]
                if( visited[node3] == False ):
                    pq.put((weight , (node2,node3,weight) ))
        if( tree.getEdges() != self.nodes-1 ):
            print("Error: Graph is not connected, it is impossible to build a MST")
            exit()
        return tree
```

Usa el algoritmo de Prim para calcular el MST. Este tiene complejidad de O(M log M), siendo M la cantidad de arcos del grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda DFS que inicia desde el nodo.

```
NDWGraph grafo [8];
...
minSpanningTree = grafo.getMinimumSpanningTree();
```

n. getMaximumSpanningTree

Función que calcula un árbol de máxima expansión de un grafo conexo, es decir, el subconjunto de aristas del grafo tal que formen un árbol con todos los nodos del grafo y la suma de sus pesos sea máxima.

Implementación en Python

No recibe argumentos. Retorna un grafo no dirigido con pesos, un árbol de mínima expansión del grafo actual.

```
def MaximumSpanningTree( self ):
        tree = UndirectedWeightedGraph( self.nodes )
        pq = PriorityQueue()
        visited = [ False ] * self.nodes
        visited[0] = True
        for edge in self.adjacencyList[0]:
            node = edge[0]
            weight = edge[1]
            if( visited[node] == False ):
                pq.put((-weight , (0,node,weight) ))
        while( pq.empty() == False ):
            currentEdge = pq.get()
            node1 = currentEdge[1][0]
            node2 = currentEdge[1][1]
            weight = currentEdge[1][2]
            if ( visited[node2] == True ):
                continue
            visited[node2] = True
            tree.addEdge( node1 , node2 , weight )
            for edge in self.adjacencyList[node2]:
                node3 = edge[0]
                weight = edge[1]
                if( visited[node3] == False ):
```

```
                    pq.put((weight , (node2,node3,weight) ))
        if( tree.getEdges() != self.nodes-1 ):
            print("Error: Graph is not connected, it is impossible to build a MST")
            exit()
        return tree
```

Usa el algoritmo de Prim para calcular el MST, modificado para darle mayor prioridad a los arcos con mayor peso. Este tiene complejidad de O(M log M), siendo M la cantidad de arcos del grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda DFS que inicia desde el nodo.

```
NDWGraph grafo [8];
...
maxSpanningTree = grafo.getMaximumSpanningTree();
```

3. Grafo dirigido sin pesos

Los grafos dirigidos sin pesos son modelados como una clase de python, cuyo constructor es el siguiente:

```
class DirectedGraph:
    def __init__( self , numberOfNodes ):
        if( numberOfNodes <= 0 ):
            print("Error: Number of nodes must be positive")
            exit()
        self.nodes = numberOfNodes
        self.adjacencyList = []
        self.numberOfEdges = 0
        self.adjacencyMatrix = []
        for i in range(numberOfNodes):
```

```
            self.adjacencyMatrix.append( [None] * self.nodes )
        for i in range(numberOfNodes):
            self.adjacencyList.append( [] )
```

Dada la creación de la matriz de adyacencia vacía inicial, la complejidad del constructor es O(N^2) siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

Recibe como único parámetro la cantidad de nodos del grafo a crear. Por defecto, sus nodos se numeran de 0 a N-1, siendo N la cantidad de nodos del grafo.

```
DGraph grafo [8];
```

a. addEdge

Función que permite agregar un arco al grafo.

Implementación en Python

Recibe dos argumentos, que deben ser índices válidos de nodos del grafo. No permite arcos repetidos, ni arcos que empiecen y terminen en el mismo nodo. Retorna un valor booleano que representa si el arco fue creado con éxito.

```
def addEdge( self , node1 , node2 ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        if( node1 == node2 ):
            print("Error: Self loops are not allowed")
            exit()
        if( self.adjacencyMatrix[node1][node2] != None ):
            print("Error: Edge between",node1,"and",node2,"already exists")
```

```
            exit()
        self.numberOfEdges = self.numberOfEdges + 1
        self.adjacencyMatrix[node1][node2] = 1
        self.adjacencyList[node1].append( node2 )
        return True
```

Tiene complejidad O(1).

Utilización en EZGraph

Recibe como parámetros los índices de los nodos que serán conectados por un nuevo arco unidireccional.

```
DGraph grafo [8];
grafo.addEdge(0,5);
```

b. deleteEdge

Función que permite eliminar un arco del grafo.

Implementación en Python

Recibe dos argumentos, que deben ser índices válidos de nodos del grafo. Retorna un valor booleano que representa si el arco fue eliminado con éxito.

```
def deleteEdge(self, node1, node2 ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        for edge in self.adjacencyList[node1]:
            if edge == node2:
                self.adjacencyList[node1].remove( edge )
```

```
                self.adjacencyMatrix[node1][node2] = None
                self.numberOfEdges = self.numberOfEdges - 1
                return True
        print("Error: Edge between",node1,"and",node2,"does not exist")
        exit()
```

Tiene complejidad O(M), siendo M la cantidad de arcos del grafo, dado que se debe recorrer la lista de adyacencia para encontrar el arco y después se elimina de una lista estándar de Python.

Utilización en EZGraph

Recibe como parámetros los índices de los nodos del arco bidireccional que será eliminado del grafo. Los índices pueden ser dados en cualquier orden.

```
DGraph grafo [8];
grafo.addEdge(0,5);
grafo.deleteEdge(0,5);
```

c. getNumEdges

Función que retorna la cantidad de arcos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNumEdges( self ):
        return self.numberOfEdges
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de arcos en el grafo.

```
DGraph grafo [8];
...
edges = grafo.getNumEdges();
```

d. getNodes

Función que retorna la cantidad de nodos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNodes( self ):
        return self.nodes
```

Este valor se mantiene constante en el grafo. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de nodos en el grafo.

```
DGraph grafo [8];
nodes = grafo.getNodes();
```

e. getEdges

Función que retorna una lista con todos los arcos del grafo

Implementación en Python

No recibe ningún argumento. Retorna una lista con todos los arcos actuales del grafo.

```
def getEdges( self ):
        allEdges = []
        for node1 in range(self.nodes):
            for node2 in self.adjacencyList[node1]:
                if( node1 < node2 ):
                    allEdges.append( [node1,node2] )
        return allEdges
```

Calcula la lista recorriendo la lista de adyacencia del grafo. Tiene complejidad O(N+M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

No recibe ningún argumento. Retorna una lista de listas de tamaño 2, todos los arcos del grafo.

```
DGraph grafo [8];
...
edges = grafo.getEdges();
```

f. getMatrix

Función que retorna la matriz de adyacencia del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getAdjacencyMatrix( self ):
        return self.adjacencyMatrix
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1). Sus valores posibles para una posición i,j son None sí no existe arco entre el nodo i y el nodo j y 1 sí existe un arco entre estos nodos.

Utilización en EZGraph

No recibe ningún argumento. Retorna una matriz, la matriz de adyacencia del grafo.

```
DGraph grafo [8];
nodes = grafo.getMatrix();
```

g. getDistanceFromNode

Función que retorna la distancia desde un nodo específico a todos los nodos.

Implementación en Python

Recibe un argumento, que debe ser un índice válido de un nodo del grafo. Retorna una lista con la distancia desde el nodo inicial dado a todos los nodos del grafo. Sí no es posible llegar algún nodo desde el nodo inicial, retorna infinito.

```
def minDistanceFromSourceToAll( self , source ):
        if( source < 0 or source >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        q = queue.Queue()
        q.put( source )
        while( q.empty() == False ):
            currentNode = q.get()
```

```
            for nextNode in self.adjacencyList[currentNode]:
                if( distance[nextNode] > distance[currentNode] + 1 ):
                    distance[nextNode] = distance[currentNode] + 1
                    q.put( nextNode )
        return distance
```

Utiliza una BFS para calcular las distancias, a partir de la estructura de datos Queue, que es una cola FIFO. Su complejidad es de O(N+M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicial. Retorna una lista de N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias del nodo inicial al nodo i-ésimo.

```
DGraph grafo [8];
...
distances = grafo.getDistanceFromNode(3);
```

h. getAllDistances

Función que retorna una matriz con las distancias entre todos los pares de nodos.

Implementación en Python

No recibe ningún argumento. Retorna una matriz con las distancias desde todos los nodos a todos los nodos del grafo. Sí no es posible llegar algún nodo desde otro, retorna infinito.

```
def minDistanceFromAllToAll( self ):
        distance = [[float('inf') for col in range(self.nodes)] for row in range(self.nodes)]
        for node1 in range( self.nodes ):
            distance[node1][node1] = 0
            for edge in self.adjacencyList[node1]:
```

```
                node2 = edge
                distance[node1][node2] = min( distance[node1][node2] , 1 )
        for k in range( self.nodes ):
            for i in range( self.nodes ):
                for j in range( self.nodes ):
                    distance[i][j] = min( distance[i][j] , distance[i][k] + distance[k][j] )
        for k in range( self.nodes ):
            for i in range( self.nodes ):
                for j in range( self.nodes ):
                    if( distance[i][j] > distance[i][k] + distance[k][j] ):
                        print("Error: Negative Weight Cycle")
                        exit()
        return distance
```

Utiliza el algoritmo de Floyd Warshall, por lo tanto tiene complejidad O(N^3), siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

No recibe argumentos. Retorna una matriz de N x N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias entre cualquier par de nodos. Para los nodos i, j tales que sea imposible ir de ia j, la distancia se muestra como infinito.

```
DGraph grafo [8];
...
distances = grafo.getAllDistances();
```

i. getDistance

Función que retorna la distancia mínima entre dos nodos dados.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna un entero con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPairDistance( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = self.minDistanceFromSourceToAll( source )
        return distance[ destination ]
```

Utiliza como base getDistanceFromNode, por lo que su complejidad es la misma, O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna un entero, la distancia mínima desde el nodo inicial al nodo final. Sí no es posible llegar del nodo inicial al nodo final, retorna infinito.

```
DGraph grafo [8];
...
distances = grafo.getAllDistances();
```

j. getShortestPath

Función que retorna los nodos de un camino mínimo entre el nodo inicial y el nodo final.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna una lista que representa un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPath( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        q = queue.Queue()
        q.put( source )
        previousNode = [-1] * self.nodes
        while( q.empty() == False ):
            currentNode = q.get()
            for nextNode in self.adjacencyList[currentNode]:
                if( distance[nextNode] > distance[currentNode] + 1 ):
                    distance[nextNode] = distance[currentNode] + 1
                    previousNode[nextNode] = currentNode
                    q.put( nextNode )
        if( previousNode[destination] == -1 ):
            print( "Error: No path between the nodes" )
            exit()
        path = deque([destination])
        currentNode = destination
        while( currentNode != source ):
            currentNode = previousNode[currentNode]
            path.appendleft(currentNode)
        return list(path)
```

Utiliza una BFS que lleva además el registro del nodo inmediatamente anterior para cada nodo. Con esta información, al final se reconstruye el camino mínimo y se retorna la lista. Su complejidad es O(N+M) dado el uso de una BFS con una cola FIFO, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna una lista, un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
DGraph grafo [8];
...
path = grafo.getShortestPath(6,1);
```

k. BFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda BFS, también conocida como búsqueda a lo ancho.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda BFS.

```
def BFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        q = queue.Queue()
        visited[node] = True
        q.put( node )
        while( q.empty() == False ):
            currNode = q.get()
            order.append( currNode )
            for nextNode in self.adjacencyList[currNode]:
                if visited[nextNode] == False:
                    visited[nextNode] = True
                    q.put( nextNode )
        return order
```

Utiliza una BFS que lleva además el registro de los nodos visitados. Su complejidad es O(N+M) dado el uso de una BFS con una cola FIFO, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda BFS que inicia desde el nodo.

```
DGraph grafo [8];
...
BFSorder = grafo.BFS(7);
```

I. DFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda DFS , también conocida como búsqueda en profundidad.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda DFS.

```
def DFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        def DFSrecursion( node ):
            nonlocal order
            nonlocal visited
            order.append( node )
```

```
            visited[node] = True
            for nextNode in self.adjacencyList[node]:
                if visited[nextNode] == False:
                    DFSrecursion( nextNode )
        DFSrecursion( node )
        return order
```

Utiliza recursión para visitar los nodos primero en profundidad. Visita cada nodo una vez y cada arista la considera una vez, por lo que su complejidad es de O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda DFS que inicia desde el nodo.

```
DGraph grafo [8];
...
DFSorder = grafo.DFS(0);
```

m. hasCycle

Función que indica si el grafo dirigido contiene un ciclo, o sí es un grafo dirigido acíclico (DAG).

Implementación en Python

No recibe argumentos. Retorna verdadero si el grafo dirigido contiene algún ciclo, y falso sí no contiene ningún ciclo.

```
def hasCycle( self ):
        visited = [False] * self.nodes
        inStack = [False] * self.nodes
        cycle = False
```

```
        def CycleDFS( node ):
            nonlocal visited
            nonlocal inStack
            nonlocal cycle
            if visited[node] == True:
                return
            visited[node] = True
            inStack[node] = True
            for nextNode in self.adjacencyList[node]:
                if inStack[nextNode] == True:
                    cycle = True
                elif visited[nextNode] == False:
                    CycleDFS( nextNode )
            inStack[node] = False
        for node in range( self.nodes ):
            CycleDFS( node )
        return cycle
```

Utiliza recursiones de DFS para determinar si existe algún back edge. Su complejidad es de O(N+M) dada la DFS, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

No recibe argumentos. Retorna verdadero si el grafo dirigido contiene algún ciclo, y falso sí no contiene ningún ciclo.

```
DGraph grafo [8];
...
cycle = grafo.hasCycle();
```

n. getSCC

Función que realiza la descomposición del grafo dirigido en sus componentes fuertemente conexas (SCC). Una componente fuertemente conexa es un conjunto de nodos tal que se puede llegar de cualquier nodo del conjunto a cualquier otro nodo del conjunto.

Implementación en Python

No recibe argumentos. Retorna una lista con N elementos, siendo N la cantidad de nodos en el grafo. La lista en la posición i-ésima contiene el identificador de la componente fuertemente conexa a la que cada nodo pertenece (un entero entre 0 y N, que es la cantidad máxima de componentes).

```
def SCC( self ):
        disc = [-1] * self.nodes
        low = [-1] * self.nodes
        components = [-1] * self.nodes
        inStack = [ False ] * self.nodes
        stack = LifoQueue( self.nodes )
        nextComponent = 0
        nextDiscovery = 0

        def tarjanDFS( node ):
            nonlocal disc
            nonlocal low
            nonlocal components
            nonlocal inStack
            nonlocal stack
            nonlocal nextComponent
            nonlocal nextDiscovery
            if( disc[node] != -1 ):
                return
            disc[node] = nextDiscovery
            low[node] = disc[node]
            nextDiscovery = nextDiscovery + 1
            inStack[node] = True
            stack.put( node )
            for nextNode in self.adjacencyList[node]:
                if( disc[nextNode] == -1 ):
                    tarjanDFS( nextNode )
```

```
                    low[node] = min( low[node] , low[nextNode] )
                elif( inStack[nextNode] == True ):
                    low[node] = min( low[node] , disc[nextNode] )
            if( disc[node] == low[node] ):
                while( True ):
                    currNode = stack.get()
                    components[currNode] = nextComponent
                    inStack[currNode] = False
                    if( currNode == node ):
                        break
                nextComponent = nextComponent + 1

        for node in range( self.nodes ):
            tarjanDFS( node )
        return components
```

Utiliza el algoritmo de Tarjan para calcular las componentes fuertemente conexas. Este se basa en un recorrido DFS del grafo, por lo que su complejidad es de O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

No recibe argumentos. Retorna una lista con el identificador de la componente a la que pertenece cada nodo. Cada identificador es un entero entre 0 y N, que es la cantidad máxima de componentes, siendo N la cantidad de nodos en el grafo.

```
DGraph grafo [8];
...
components = grafo.getSCC();
```

o. getTopologicalOrder

Función que retorna un posible orden topológico para el grafo dirigido. Un orden topológico de un grafo dirigido es un ordenamiento de los nodos tal que, si se toman los arcos como prerrequisitos entre los nodos, ningún nodo aparece en el orden antes que alguno de sus prerrequisitos.

Implementación en Python

No recibe argumentos. Retorna una lista con N elementos, siendo N la cantidad de nodos en el grafo, un posible orden topológico para el grafo. Da un error sí en el grafo existe un ciclo, dado que en este caso es imposible construir un orden topológico que incluya todos los nodos.

```
def topSort ( self ):
        q = queue.Queue()
        inDegree = [0] * self.nodes
        for node1 in range( self.nodes ):
            for node2 in self.adjacencyList[node1]:
                inDegree[node2] = inDegree[node2]+1
        topOrder = []
        for node in range( self.nodes ):
            if( inDegree[node] == 0 ):
                q.put( node )
        while( q.empty() == False ):
            currNode = q.get()
            topOrder.append( currNode )
            for nextNode in self.adjacencyList[currNode]:
                inDegree[nextNode] = inDegree[nextNode]-1
                if( inDegree[nextNode] == 0 ):
                    q.put( nextNode )
        if( len(topOrder) != self.nodes ):
            print("Error: Not possible to construct a topological sort")
            exit()
        return topOrder
```

Utilización en EZGraph

No recibe argumentos. Retorna una lista con N elementos, siendo N la cantidad de nodos en el grafo, un posible orden topológico para el grafo.

```
DGraph grafo [8];
...
```

```
topologicalOrder = grafo.getTopologicalOrder();
```

4. Grafo dirigido con pesos

Los grafos dirigidos con pesos son modelados como una clase de python, cuyo constructor es el siguiente:

```
class DirectedWeightedGraph:
    def __init__( self , numberOfNodes ):
        if( numberOfNodes <= 0 ):
            print("Error: Number of nodes must be positive")
            exit()
        self.nodes = numberOfNodes
        self.adjacencyList = []
        self.numberOfEdges = 0
        self.adjacencyMatrix = []
        for i in range(numberOfNodes):
            self.adjacencyMatrix.append( [None] * self.nodes )
        for i in range(numberOfNodes):
            self.adjacencyList.append( [] )
```

Dada la creación de la matriz de adyacencia vacía inicial, la complejidad del constructor es O(N^2) siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

Recibe como único parámetro la cantidad de nodos del grafo a crear. Por defecto, sus nodos se numeran de 0 a N-1, siendo N la cantidad de nodos del grafo.

```
DWGraph grafo [8];
```

a. addEdge

Función que permite agregar un arco al grafo.

Implementación en Python

Recibe tres argumentos, que deben ser índices válidos de nodos del grafo, y un peso que puede ser entero o decimal. No permite arcos repetidos, ni arcos que empiecen y terminen en el mismo nodo. Retorna un valor booleano que representa si el arco fue creado con éxito.

```
def addEdge( self , node1 , node2 , weight ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        if( node1 == node2 ):
            print("Error: Self loops are not allowed")
            exit()
        if( self.adjacencyMatrix[node1][node2] != None ):
            print("Error: Edge between",node1,"and",node2,"already exists")
            exit()
        self.numberOfEdges = self.numberOfEdges + 1
        self.adjacencyMatrix[node1][node2] = weight
        self.adjacencyList[node1].append( [node2 , weight] )
        return True
```

Tiene complejidad O(1).

Utilización en EZGraph

Recibe como parámetros los índices de los nodos que serán conectados por un nuevo arco unidireccional, y el peso del arco que puede ser entero o decimal.

```
DWGraph grafo [8];
grafo.addEdge(0,5,-1.5);
```

b. deleteEdge

Función que permite eliminar un arco del grafo.

Implementación en Python

Recibe dos argumentos, que deben ser índices válidos de nodos del grafo. Retorna un valor booleano que representa si el arco fue eliminado con éxito.

```
def deleteEdge(self, node1, node2 ):
        if( node1 < 0 or node1 >= self.nodes or node2 < 0 or node2 >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        for edge in self.adjacencyList[node1]:
            if edge[0] == node2:
                self.adjacencyList[node1].remove( edge )
                self.adjacencyMatrix[node1][node2] = None
                self.numberOfEdges = self.numberOfEdges - 1
                return True
        return False
```

Tiene complejidad O(M), siendo M la cantidad de arcos del grafo, dado que se debe recorrer la lista de adyacencia para encontrar el arco y después se elimina de una lista estándar de Python.

Utilización en EZGraph

Recibe como parámetros los índices de los nodos del arco bidireccional que será eliminado del grafo. Los índices pueden ser dados en cualquier orden.

```
DWGraph grafo [8];
```

```
grafo.addEdge(0,5);
grafo.deleteEdge(0,5);
```

c. getNumEdges

Función que retorna la cantidad de arcos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNumEdges( self ):
        return self.numberOfEdges
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de arcos en el grafo.

```
DWGraph grafo [8];
...
edges = grafo.getNumEdges();
```

d. getNodes

Función que retorna la cantidad de nodos del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getNodes( self ):
        return self.nodes
```

Este valor se mantiene constante en el grafo. Se puede obtener en O(1).

Utilización en EZGraph

No recibe ningún argumento. Retorna un entero, la cantidad de nodos en el grafo.

```
DWGraph grafo [8];
nodes = grafo.getNodes();
```

e. getEdges

Función que retorna una lista con todos los arcos del grafo

Implementación en Python

No recibe ningún argumento. Retorna una lista con todos los arcos actuales del grafo.

```
def getEdges( self ):
        allEdges = []
        for node1 in range(self.nodes):
            for edge in self.adjacencyList[node1]:
                node2 = edge[0]
                weight = edge[1]
                allEdges.append( [node1,node2,weight] )
        return allEdges
```

Calcula la lista recorriendo la lista de adyacencia del grafo. Tiene complejidad O(N+M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

No recibe ningún argumento. Retorna una lista de listas de tamaño 2, todos los arcos del grafo.

```
DWGraph grafo [8];
...
edges = grafo.getEdges();
```

f. getMatrix

Función que retorna la matriz de adyacencia del grafo.

Implementación en Python

Se obtiene del valor del atributo de la clase, al que se accede utilizando la función:

```
def getAdjacencyMatrix( self ):
        return self.adjacencyMatrix
```

Este valor se va actualizando a medida que se crean y eliminan arcos. Se puede obtener en O(1). Sus valores posibles para una posición i,j son None sí no existe arco entre el nodo i y el nodo j y 1 sí existe un arco entre estos nodos.

Utilización en EZGraph

No recibe ningún argumento. Retorna una matriz, la matriz de adyacencia del grafo.

```
DWGraph grafo [8];
nodes = grafo.getMatrix();
```

g. getDistanceFromNode

Función que retorna la distancia desde un nodo específico a todos los nodos.

Implementación en Python

Recibe un argumento, que debe ser un índice válido de un nodo del grafo. Retorna una lista con la distancia desde el nodo inicial dado a todos los nodos del grafo. Sí no es posible llegar algún nodo desde el nodo inicial, retorna infinito.

```
def minDistanceFromSourceToAll( self , source ):
        if( source < 0 or source >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        for i in range( self.nodes-1 ):
            for node1 in range( self.nodes ):
                for edge in self.adjacencyList[node1]:
                    node2 = edge[0]
                    weight = edge[1]
                    if( distance[node2] > distance[node1] + weight ):
                        distance[node2] = distance[node1] + weight
        for node1 in range( self.nodes ):
            for edge in self.adjacencyList[node1]:
                node2 = edge[0]
                weight = edge[1]
                if( distance[node2] > distance[node1] + weight ):
                    print("Error: Negative Weight Cycle")
                    exit()
        return distance
```

Utiliza el algoritmo de Bellman Ford para calcular las distancias, utilizando ciclos anidados para relajar cada arco N-1 veces. Su complejidad es de O(N*M), siendo N la cantidad de nodos del grafo, y M la cantidad de arcos del grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicial. Retorna una lista de N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias del nodo inicial al nodo i-ésimo.

```
DWGraph grafo [8];
...
distances = grafo.getDistanceFromNode(3);
```

h. getAllDistances

Función que retorna una matriz con las distancias entre todos los pares de nodos.

Implementación en Python

No recibe ningún argumento. Retorna una matriz con las distancias desde todos los nodos a todos los nodos del grafo. Sí no es posible llegar algún nodo desde otro, retorna infinito.

```
def minDistanceFromAllToAll( self ):
        distance = [[float('inf') for col in range(self.nodes)] for row in range(self.nodes)]
        for node1 in range( self.nodes ):
            distance[node1][node1] = 0
            for edge in self.adjacencyList[node1]:
                node2 = edge
                distance[node1][node2] = min( distance[node1][node2] , 1 )
        for k in range( self.nodes ):
            for i in range( self.nodes ):
                for j in range( self.nodes ):
                    distance[i][j] = min( distance[i][j] , distance[i][k] + distance[k][j] )
        for k in range( self.nodes ):
            for i in range( self.nodes ):
```

```
                for j in range( self.nodes ):
                    if( distance[i][j] > distance[i][k] + distance[k][j] ):
                        print("Error: Negative Weight Cycle")
                        exit()
        return distance
```

Utiliza el algoritmo de Floyd Warshall, por lo tanto tiene complejidad O(N^3), siendo N la cantidad de nodos del grafo.

Utilización en EZGraph

No recibe argumentos. Retorna una matriz de N x N elementos, siendo N la cantidad de nodos en el grafo, que contiene las distancias entre cualquier par de nodos. Para los nodos i, j tales que sea imposible ir de ia j, la distancia se muestra como infinito.

```
DWGraph grafo [8];
...
distances = grafo.getAllDistances();
```

i. getDistance

Función que retorna la distancia mínima entre dos nodos dados.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna un entero con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPairDistance( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = self.minDistanceFromSourceToAll( source )
        return distance[ destination ]
```

Utiliza como base getDistanceFromNode, por lo que su complejidad es la misma, O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna un entero, la distancia mínima desde el nodo inicial al nodo final. Sí no es posible llegar del nodo inicial al nodo final, retorna infinito.

```
DWGraph grafo [8];
...
distances = grafo.getAllDistances();
```

j. getShortestPath

Función que retorna los nodos de un camino mínimo entre el nodo inicial y el nodo final.

Implementación en Python

Recibe como argumento los índices de los nodos. Retorna una lista que representa un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
def minPath( self , source , destination ):
        if( source < 0 or source >= self.nodes or destination < 0 or destination >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        distance = [ float('inf') ] * self.nodes
        distance[source] = 0
        previousNode = [-1] * self.nodes
        for i in range( self.nodes-1 ):
            for node1 in range( self.nodes ):
                for edge in self.adjacencyList[node1]:
                    node2 = edge[0]
```

```
                    weight = edge[1]
                    if( distance[node2] > distance[node1] + weight ):
                        distance[node2] = distance[node1] + weight
                        previousNode[node2] = node1
        for node1 in range( self.nodes ):
            for edge in self.adjacencyList[node1]:
                node2 = edge[0]
                weight = edge[1]
                if( distance[node2] > distance[node1] + weight ):
                    print("Error: Negative Weight Cycle")
                    exit()
        if( previousNode[destination] == -1 ):
            print("Error: No path between the nodes")
            exit()
        path = deque([destination])
        currentNode = destination
        while( currentNode != source ):
            currentNode = previousNode[currentNode]
            path.appendleft(currentNode)
        return list(path)
```

Utiliza el algoritmo de Bellman Ford, pero además lleva además el registro del nodo inmediatamente anterior para cada nodo. Con esta información, al final se reconstruye el camino mínimo y se retorna la lista. Su complejidad es O(N*M) dado el uso de Bellman Ford, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe dos argumentos, los índices del nodo inicial y del nodo final. Retorna una lista, un camino en el grafo con la distancia mínima de ir desde el primer nodo dado hasta el segundo nodo dado.

```
DWGraph grafo [8];
...
path = grafo.getShortestPath(6,1);
```

k. BFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda BFS, también conocida como búsqueda a lo ancho.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda BFS.

```
def BFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        q = queue.Queue()
        visited[node] = True
        q.put( node )
        while( q.empty() == False ):
            currNode = q.get()
            order.append( currNode )
            for edge in self.adjacencyList[currNode]:
                nextNode = edge[0]
                if visited[nextNode] == False:
                    visited[nextNode] = True
                    q.put( nextNode )
        return order
```

Utiliza una BFS que lleva además el registro de los nodos visitados. Su complejidad es O(N+M) dado el uso de una BFS con una cola FIFO, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda BFS que inicia desde el nodo.

```
DWGraph grafo [8];
...
BFSorder = grafo.BFS(7);
```

I. DFS

Función que retorna los nodos que pueden ser visitados desde un nodo inicial en el orden de la búsqueda DFS , también conocida como búsqueda en profundidad.

Implementación en Python

Recibe un argumento, el índice del nodo inicial. Retorna una lista con los nodos a los que se puede llegar desde el nodo inicial, en el orden en el que son visitados en la búsqueda DFS.

```
def DFS( self , node ):
        if( node < 0 or node >= self.nodes ):
            print("Error: Node index is out of bounds. Valid indexes are between 0 and",self.nodes)
            exit()
        order = []
        visited = [False] * self.nodes
        def DFSrecursion( node ):
            nonlocal order
            nonlocal visited
            order.append( node )
            visited[node] = True
            for edge in self.adjacencyList[node]:
                nextNode = edge[0]
```

```
                if visited[nextNode] == False:
                    DFSrecursion( nextNode )
        DFSrecursion( node )
        return order
```

Utiliza recursión para visitar los nodos primero en profundidad. Visita cada nodo una vez y cada arista la considera una vez, por lo que su complejidad es de O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

Recibe un argumento, el índice del nodo inicia. Retorna una lista, el orden en el que son visitados los nodos en la búsqueda DFS que inicia desde el nodo.

```
DWGraph grafo [8];
...
DFSorder = grafo.DFS(0);
```

m. hasCycle

Función que indica si el grafo dirigido contiene un ciclo, o sí es un grafo dirigido acíclico (DAG).

Implementación en Python

No recibe argumentos. Retorna verdadero si el grafo dirigido contiene algún ciclo, y falso sí no contiene ningún ciclo.

```
def hasCycle( self ):
        visited = [False] * self.nodes
        inStack = [False] * self.nodes
        cycle = False

        def CycleDFS( node ):
            nonlocal visited
```

```
            nonlocal inStack
            nonlocal cycle
            if visited[node] == True:
                return
            visited[node] = True
            inStack[node] = True
            for edge in self.adjacencyList[node]:
                nextNode = edge[0]
                if inStack[nextNode] == True:
                    cycle = True
                elif visited[nextNode] == False:
                    CycleDFS( nextNode )
            inStack[node] = False
        for node in range( self.nodes ):
            CycleDFS( node )
        return cycle
```

Utiliza recursiones de DFS para determinar si existe algún back edge. Su complejidad es de O(N+M) dada la DFS, siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

No recibe argumentos. Retorna verdadero si el grafo dirigido contiene algún ciclo, y falso sí no contiene ningún ciclo.

```
DWGraph grafo [8];
...
cycle = grafo.hasCycle();
```

n. getSCC

Función que realiza la descomposición del grafo dirigido en sus componentes fuertemente conexas (SCC). Una componente fuertemente conexa es un conjunto de nodos tal que se puede llegar de cualquier nodo del conjunto a cualquier otro nodo del conjunto.

Implementación en Python

No recibe argumentos. Retorna una lista con N elementos, siendo N la cantidad de nodos en el grafo. La lista en la posición i-ésima contiene el identificador de la componente fuertemente conexa a la que cada nodo pertenece (un entero entre 0 y N, que es la cantidad máxima de componentes).

```
def SCC( self ):
        disc = [-1] * self.nodes
        low = [-1] * self.nodes
        components = [-1] * self.nodes
        inStack = [ False ] * self.nodes
        stack = LifoQueue( self.nodes )
        nextComponent = 0
        nextDiscovery = 0

        def tarjanDFS( node ):
            nonlocal disc
            nonlocal low
            nonlocal components
            nonlocal inStack
            nonlocal stack
            nonlocal nextComponent
            nonlocal nextDiscovery
            if( disc[node] != -1 ):
                return
            disc[node] = nextDiscovery
            low[node] = disc[node]
            nextDiscovery = nextDiscovery + 1
            inStack[node] = True
            stack.put( node )
            for edge in self.adjacencyList[node]:
                nextNode = edge[0]
                if( disc[nextNode] == -1 ):
```

```
                    tarjanDFS( nextNode )
                    low[node] = min( low[node] , low[nextNode] )
                elif( inStack[nextNode] == True ):
                    low[node] = min( low[node] , disc[nextNode] )
            if( disc[node] == low[node] ):
                while( True ):
                    currNode = stack.get()
                    components[currNode] = nextComponent
                    inStack[currNode] = False
                    if( currNode == node ):
                        break
                nextComponent = nextComponent + 1

        for node in range( self.nodes ):
            tarjanDFS( node )
        return components
```

Utiliza el algoritmo de Tarjan para calcular las componentes fuertemente conexas. Este se basa en un recorrido DFS del grafo, por lo que su complejidad es de O(N+M), siendo N la cantidad de nodos en el grafo y M la cantidad de arcos en el grafo.

Utilización en EZGraph

No recibe argumentos. Retorna una lista con el identificador de la componente a la que pertenece cada nodo. Cada identificador es un entero entre 0 y N, que es la cantidad máxima de componentes, siendo N la cantidad de nodos en el grafo.

```
DWGraph grafo [8];
...
components = grafo.getSCC();
```

o. getTopologicalOrder

Función que retorna un posible orden topológico para el grafo dirigido. Un orden topológico de un grafo dirigido es un ordenamiento de los nodos tal que, si se toman los arcos como prerrequisitos entre los nodos, ningún nodo aparece en el orden antes que alguno de sus prerrequisitos.

Implementación en Python

No recibe argumentos. Retorna una lista con N elementos, siendo N la cantidad de nodos en el grafo, un posible orden topológico para el grafo. Da un error sí en el grafo existe un ciclo, dado que en este caso es imposible construir un orden topológico que incluya todos los nodos.

```
def topSort ( self ):
        q = queue.Queue()
        inDegree = [0] * self.nodes
        for node1 in range( self.nodes ):
            for edge in self.adjacencyList[node1]:
                node2 = edge[0]
                weight = edge[1]
                inDegree[node2] = inDegree[node2]+1
        topOrder = []
        for node in range( self.nodes ):
            if( inDegree[node] == 0 ):
                q.put( node )
        while( q.empty() == False ):
            currNode = q.get()
            topOrder.append( currNode )
            for edge in self.adjacencyList[currNode]:
                nextNode = edge[0]
                inDegree[nextNode] = inDegree[nextNode]-1
                if( inDegree[nextNode] == 0 ):
                    q.put( nextNode )
        if( len(topOrder) != self.nodes ):
            print("Error: Not possible to construct a topological sort")
            exit()
        return topOrder
```

Utilización en EZGraph

No recibe argumentos. Retorna una lista con N elementos, siendo N la cantidad de nodos en el grafo, un posible orden topológico para el grafo.

```
DWGraph grafo [8];
...
topologicalOrder = grafo.getTopologicalOrder();
```

5. Paint

Función que pinta el grafo actual en una ventana nueva.

Utilización en EZGraph

No recibe argumentos. Abre una ventana nueva con una visualización del grafo. Funciona con cualquier tipo de grafo.

```
NDGraph grafo [8];
...
grafo.paint();
```

6. Lectura

Permite al usuario leer el valor de una variable por consola. Permite ingresar valores enteros, decimales, y cadenas de texto entre comillas dobles.

Utilización en EZGraph

Recibe un argumento, la variable donde se debe almacenar el valor ingresado.

```
read(a)
```

7. Escritura

Permite al usuario imprimir un valor por consola.

Utilización en EZGraph

Recibe un argumento, la variable o valor a ser impreso.

```
print(a)
```

8. Ciclos

Permiten al usuario repetir acciones un cierto número de veces.

Utilización en EZGraph

Recibe un argumento, la variable o valor a ser impreso.

```
for i = 1:5{
      // instrucciones a repetir
}
```