Algoritmo y Estructura de Datos II

TP Grafos Parte I y II

Alumno:Gonzalez Sanchez Gabriel
Legajo: 12007

Ejercicio 1

Implementar la función crear grafo que dada una lista de vértices y una lista de aristas cree un grafo con la representación por Lista de Adyacencia.

def createGraph(List, List)

Descripción: Implementa la operación crear grafo
Entrada: LinkedList con la lista de vértices y **LinkedList** con la lista de aristas donde por cada par de elementos representa una conexión entre dos vértices.
Salida: retorna el nuevo grafo

```
#Ejercicio 1
#Crea un grafo representandolo por lista de adyacencia
#Retorna el grafo
def createGraph(List_vertice,List_aristas):
    Graph = Array(length(List_vertice),vertex())

    #Inserta los vertices en el array
    currentNode = List_vertice.head
    for i in range(0,len(Graph)):
        Graph[i] = vertex()
        Graph[i].value = currentNode.value
        Graph[i].ady = LinkedList()
        currentNode = currentNode.nextNode

    #Inserta los adyacentes de cada vertice
    currentNode = List_aristas.head
    while currentNode.nextNode.nextNode!=None:
        insertGraph(Graph,currentNode.value,currentNode.nextNode.value)
        currentNode=currentNode.nextNode.nextNode
    insertGraph(Graph,currentNode.value,currentNode.nextNode.value)

    return Graph
```

```python
#Inserta una arista entre verA y verB
#Utiliza lista de adyacencia
def insertGraph(Graph,verA,verB):
    n = len(Graph)
    for i in range(0,n):
        if Graph[i].value == verA:
            for j in range(0,n):
                if Graph[j].value == verB:
                    add(Graph[i].ady,Graph[j])
        if Graph[i].value == verB:
            for j in range(0,n):
                if Graph[j].value == verA:
                    add(Graph[i].ady,Graph[j])
```

Ejercicio 2

Implementar la función que responde a la siguiente especificación.

def existPath(Grafo, v1, v2):

Descripción: Implementa la operación existe camino que busca si existe un camino entre los vértices v1 y v2

Entrada: Grafo con la representación de Lista de Adyacencia, **v1** y **v2** vértices en el grafo.

Salida: retorna **True** si existe camino entre v1 y v2, **False** en caso contrario.

```
#Ejercicio 2
#Determina si existe un camino entre dos vertices
#Utiliza DFS para encontrar un camino
#Devuelve True si existe y False si no
def existPath(G,v1,v2):
    v1.color = "gray"

    Q = LinkedList()

    enqueue(Q,v1)
    while Q.head!=None:
        u = dequeue(Q)
        currentNode = u.ady.head
        while currentNode!=None:
            if currentNode.value.color == "white":
                if currentNode.value == v2:
                    return True
                currentNode.value.color = "gray"
                enqueue(Q,currentNode.value)
            currentNode = currentNode.nextNode
        u.color = "black"
    return False
```

Ejercicio 3

Implementar la función que responde a la siguiente especificación.

def isConnected(Grafo):

Descripción: Implementa la operación es conexo

Entrada: Grafo con la representación de Lista de Adyacencia.

Salida: retorna **True** si existe camino entre todo par de vértices, **False** en caso contrario.

```
#Ejercicio 3
#Determina si un grafo es conexo o no
#Utiliza BFS para colorear de negro aquellos nodos conectados
def isConnected(G):
    BFS(G,G[0])
    n = len(G)
    count = 0
    for i in range(0,n):
        if G[i].color=="black":
            count +=1
    return count==n
```

Ejercicio 4

Implementar la función que responde a la siguiente especificación.

def isTree(Grafo):

Descripción: Implementa la operación es árbol
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna **True** si el grafo es un árbol.

```
#Ejercicio 4
#Realiza un recorrido por BFS
#Si en algun momento el vertice que apunta tiene color gris
#Significa que existe un ciclo
#Si no posee ciclos entonces es un arbol
def isTree(G):
    G[0].color = "gray"

    Q = LinkedList()
    enqueue(Q,G[0])
    while Q.head!=None:
        u = dequeue(Q)
        currentNode = u.ady.head
        while currentNode!=None:
            if currentNode.value.color=="gray":
                return False
            elif currentNode.value.color == "white":
                currentNode.value.color="gray"
                enqueue(Q,currentNode.value)
            currentNode = currentNode.nextNode
        u.color = "black"
    return True
```

Ejercicio 5

Implementar la función que responde a la siguiente especificación.

def isComplete(Grafo):

Descripción: Implementa la operación es completo
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna **True** si el grafo es completo.

Nota: Tener en cuenta que un grafo es completo cuando existe una arista entre todo par de vértices.

Ejercicio 6

Implementar una función que dado un grafo devuelva una lista de aristas que si se eliminan el grafo se convierte en un árbol. Respetar la siguiente especificación.

def convertTree(Grafo)

Descripción: Implementa la operación es convertir a árbol
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: LinkedList de las aristas que se pueden eliminar y el grafo resultante se convierte en un árbol.

```
#Ejercicio 8

#Convierte un grafo en un arbol
#Si no es conexo retorna el grafo original
def convertToTree(G,v):

    if isConnected(G)==False:
        return False

    G = resetGraph(G)
    #Determinamos si ya era un arbol
    if isTree(G)==False:
        G = resetGraph(G)
        #Determina las aristas que se deben eliminar
        T = ListToTree(G,v)
        #Inserta los adyacentes de cada vertice
        currentNode = T.head
        while currentNode.nextNode.nextNode!=None:
            deleteGraph(G,currentNode.value,currentNode.nextNode.value)
            currentNode=currentNode.nextNode.nextNode
        deleteGraph(G,currentNode.value,currentNode.nextNode.value)

    return G
```

Parte 2

Ejercicio 7

Implementar la función que responde a la siguiente especificación.

def countConnections(Grafo):

Descripción: Implementa la operación cantidad de componentes conexas
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna el número de componentes conexas que componen el grafo.

Ejercicio 8

Implementar la función que responde a la siguiente especificación.

def convertToBFSTree(Grafo, v):

Descripción: Convierte un grafo en un árbol BFS
Entrada: Grafo con la representación de Lista de Adyacencia, **v** vértice que representa la raíz del árbol
Salida: Devuelve una Lista de Adyacencia con la representación BFS del grafo recibido usando **v** como raíz.

Ejercicio 9

Implementar la función que responde a la siguiente especificación.

def convertToDFSTree(Grafo, v):

Descripción: Convierte un grafo en un árbol DFS
Entrada: Grafo con la representación de Lista de Adyacencia, **v** vértice que representa la raíz del árbol
Salida: Devuelve una Lista de Adyacencia con la representación DFS del grafo recibido usando **v** como raíz.

Ejercicio 10

Implementar la función que responde a la siguiente especificación.

def bestRoad(Grafo, v1, v2):

Descripción: Encuentra el camino más corto, en caso de existir, entre dos vértices.
Entrada: Grafo con la representación de Lista de Adyacencia, **v1** y **v2** vértices del grafo.

Salida: retorna la lista de vértices que representan el camino más corto entre **v1** y **v2**. La lista resultante contiene al inicio a **v1** y al final a **v2.** En caso que no exista camino se retorna la lista vacía.