Trabajo Práctico
Grafos
Tomás Rando - 14004 - LCC

1)
createGraph(List1, List2)

```
#O(A + V)
def createGraph(ListA, ListB):
    n = linkedlist.length(ListA)
    Graph = Array(n, linkedlist.LinkedList())
    for i in range(0, n):
        Graph[i] = linkedlist.LinkedList()

    currentNode = ListB.head
    while currentNode != None:
        linkedlist.add(Graph[currentNode.value[0]], currentNode.value[1])
        linkedlist.add(Graph[currentNode.value[1]], currentNode.value[0])
        currentNode = currentNode.nextNode
    return Graph
```

2)
existPath(Grafo, v1, v2)

```
def existPath(Grafo, v1, v2):
    n = len(Grafo)
    for i in range(0, n):
        node = linkedlist.Node()
        node.value = "W"
        node.nextNode = Grafo[i].head
        Grafo[i].head = node
    stack = linkedlist.LinkedList()
    linkedlist.push(stack, v1)
    currentNode = stack.head
    condition = False
    while currentNode != None:
        linkedlist.pop(stack)
        Grafo[currentNode.value].head.value = "G"
        currentNode2 = Grafo[currentNode.value].head.nextNode
        while currentNode2 != None:
            if Grafo[currentNode2.value].head.value == "W":
                Grafo[currentNode2.value].head.value = "G"
                linkedlist.push(stack, currentNode2.value)
            if currentNode2.value == v2:
                condition = True
                stack.head = None
                break
            currentNode2 = currentNode2.nextNode
        Grafo[currentNode.value].head.value = "B"
        currentNode = stack.head
    for i in range(0, n):
        linkedlist.pop(Grafo[i])
    return condition
```

3)
isConnected(Grafo)

```
def isConnected(Grafo):
    return isConnectedWithBfs(Grafo)
def isConnectedWithBfs(Grafo):
    n = len(Grafo)
    for i in range(0, n):
        node = linkedlist.Node()
        node.value = "W"
        node.nextNode = Grafo[i].head
        Grafo[i].head = node
    stack = linkedlist.LinkedList()
    linkedlist.push(stack, 0)
    currentNode = stack.head
    while currentNode != None:
        linkedlist.pop(stack)
        Grafo[currentNode.value].head.value = "G"
        currentNode2 = Grafo[currentNode.value].head.nextNode
        while currentNode2 != None:
            if Grafo[currentNode2.value].head.value == "W":
                Grafo[currentNode2.value].head.value = "G"
                linkedlist.push(stack, currentNode2.value)
            currentNode2 = currentNode2.nextNode
        Grafo[currentNode.value].head.value = "B"
        currentNode = stack.head
    condition = True
    for i in range(0, n):
        if Grafo[i].head.value == "W":
            condition = False
        linkedlist.pop(Grafo[i])
    return condition
```

4)
isTree(Grafo)

```
def isTree(Grafo):
    if isConnected(Grafo) == False:
        return False
    return isTreeWithBfs(Grafo)
```

```
def isTreeWithBfs(Grafo):
    n = len(Grafo)
    for i in range(0, n):
        node = linkedlist.Node()
        node.value = "W"
        node.nextNode = Grafo[i].head
        Grafo[i].head = node
    stack = linkedlist.LinkedList()
    linkedlist.push(stack, 0)
    currentNode = stack.head
    condition = True
    while currentNode != None:
        linkedlist.pop(stack)
        Grafo[currentNode.value].head.value = "G"
        currentNode2 = Grafo[currentNode.value].head.nextNode
        while currentNode2 != None:
            if Grafo[currentNode2.value].head.value == "W":
                Grafo[currentNode2.value].head.value = "G"
                linkedlist.push(stack, currentNode2.value)
            else:
                if Grafo[currentNode2.value].head.value == "G":
                    condition = False
            currentNode2 = currentNode2.nextNode

        if condition == True:
            Grafo[currentNode.value].head.value = "B"
            currentNode = stack.head
        else:
            break
    for i in range(0, n):
        linkedlist.pop(Grafo[i])
    return condition
```

5)
isComplete(Grafo)

```
#O(V*V), podria mejorarse a O(V) con listas de python
def isComplete(Grafo):
    n = len(Grafo)
    for i in range(0, n):
        if (n-1) != linkedlist.length(Grafo[i]):
            return False
    return True
```

6)
convertTree(Grafo)

```
def convertTree(Grafo):
    return treeWithBfs(Grafo)
```

```
"""
W, G, B son abreviaciones para White, Gray y Black respectivamente. Utiliza el recorrido BFS para encontrar ciclos.
"""
def treeWithBfs(Grafo):
    auxList = linkedlist.LinkedList()
    n = len(Grafo)
    for i in range(0, n):
        node = linkedlist.Node()
        node.value = "W"
        node.nextNode = Grafo[i].head
        Grafo[i].head = node
    stack = linkedlist.LinkedList()
    linkedlist.push(stack, 0)
    currentNode = stack.head
    while currentNode != None:
        linkedlist.pop(stack)
        Grafo[currentNode.value].head.value = "G"
        currentNode2 = Grafo[currentNode.value].head.nextNode
        while currentNode2 != None:
            if Grafo[currentNode2.value].head.value == "W":
                Grafo[currentNode2.value].head.value = "G"
                linkedlist.push(stack, currentNode2.value)
            else:
                if Grafo[currentNode2.value].head.value == "G":
                    linkedlist.add(auxList, (currentNode2.value, currentNode.value))
            currentNode2 = currentNode2.nextNode
        Grafo[currentNode.value].head.value = "B"
        currentNode = stack.head
    for i in range(0, n):
        linkedlist.pop(Grafo[i])
    return auxList
```

7)
countConnections(Grafo)

```
def countConnections(Grafo):
    return countConnectionsWithBfs(Grafo)
def countConnectionsWithBfs(Grafo):
    n = len(Grafo)
    for i in range(0, n):
        node = linkedlist.Node()
        node.value = "W"
        node.nextNode = Grafo[i].head
        Grafo[i].head = node
    stack = linkedlist.LinkedList()
    linkedlist.push(stack, 0)
    currentNode = stack.head
    contador = 1
    while currentNode != None:
        linkedlist.pop(stack)
        Grafo[currentNode.value].head.value = "G"
        currentNode2 = Grafo[currentNode.value].head.nextNode
        while currentNode2 != None:
            if Grafo[currentNode2.value].head.value == "W":
                Grafo[currentNode2.value].head.value = "G"
                linkedlist.push(stack, currentNode2.value)
            currentNode2 = currentNode2.nextNode
        Grafo[currentNode.value].head.value = "B"
        currentNode = stack.head
        if currentNode == None:
            for i in range(0, n):
                if Grafo[i].head.value == "W":
                    contador += 1
                    linkedlist.push(stack, i)
                    currentNode = stack.head
                    break
    for i in range(0, n):
        linkedlist.pop(Grafo[i])
    return contador
```

8)

convertToBFSTree(Grafo, v)

```
def convertToBFSTree(Grafo, v):
    n = len(Grafo)
    newGraph = Array(n, linkedlist.LinkedList())
    for i in range(0, n):
        newGraph[i] = linkedlist.LinkedList()
        node = linkedlist.Node()
        node.value = "W"
        node.nextNode = Grafo[i].head
        Grafo[i].head = node
    stack = linkedlist.LinkedList()
    linkedlist.push(stack, v)
    currentNode = stack.head
    while currentNode != None:
        linkedlist.pop(stack)
        Grafo[currentNode.value].head.value = "G"
        currentNode2 = Grafo[currentNode.value].head.nextNode
        while currentNode2 != None:
            if Grafo[currentNode2.value].head.value == "W":
                Grafo[currentNode2.value].head.value = "G"
                linkedlist.push(stack, currentNode2.value)
                linkedlist.add(newGraph[currentNode.value], currentNode2.value)
                linkedlist.add(newGraph[currentNode2.value], currentNode.value)
            currentNode2 = currentNode2.nextNode
        Grafo[currentNode.value].head.value = "B"
        currentNode = stack.head
    for i in range(0, n):
        linkedlist.pop(Grafo[i])
    return newGraph
```

9)

convertToDFSTree(Grafo, v)

```
def convertToDFSTree(Grafo, v):
    n = len(Grafo)
    newGraph = Array(n, linkedlist.LinkedList())
    for i in range(0, n):
        newGraph[i] = linkedlist.LinkedList()
        node = linkedlist.Node()
        node.value = "W"
        node.nextNode = Grafo[i].head
        Grafo[i].head = node
    Grafo[v].head.value = "G"
    newGraph = convertToDFSTreeR(Grafo, newGraph, v, v)
    for i in range(0, n):
        if Grafo[i].head.value == "W":
            Grafo[i].head.value = "G"
            newGraph = convertToDFSTreeR(Grafo, newGraph, i, i)
    for i in range(0, n):
        linkedlist.pop(Grafo[i])
    return newGraph

def convertToDFSTreeR(Grafo, newGraph, lastVertex, v):
    currentNode = Grafo[v].head.nextNode
    while currentNode != None:
        if currentNode.value != lastVertex:
            if Grafo[currentNode.value].head.value == "W":
                Grafo[currentNode.value].head.value = "G"
                linkedlist.add(newGraph[v], currentNode.value)
                linkedlist.add(newGraph[currentNode.value], v)
                convertToDFSTreeR(Grafo, newGraph, v, currentNode.value)
        currentNode = currentNode.nextNode
    Grafo[v].head.value = "B"
    return newGraph
```

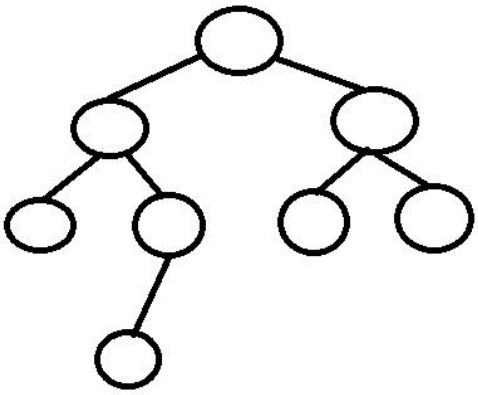
10)

bestRoad(Grafo, v1, v2)

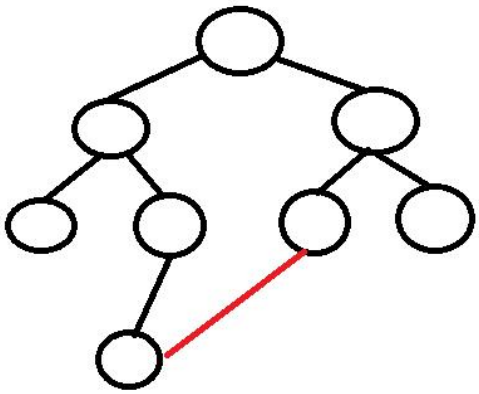
```
def bestRoad(Grafo, v1, v2):
    if (len(Grafo) < v1) or len(Grafo) < v2:
        return []
    newGraph = convertToBFSTree(Grafo, v1)
    condition = bestRoadR(newGraph, v2, [v1], v1, v1)
    if condition != False:
        return condition
    else:
        return []

def bestRoadR(Grafo, v2, auxList, lastVertex, Vertex):
    if linkedlist.search(Grafo[Vertex], v2) != None:
        auxList.append(v2)
        return auxList
    condition = False
    currentNode = Grafo[Vertex].head
    while currentNode != None:
        if currentNode.value != lastVertex:
            auxList.append(currentNode.value)
            condition = bestRoadR(Grafo, v2, auxList, Vertex, currentNode.value)
            if condition == False:
                auxList.pop()
            else:
                return condition
        currentNode = currentNode.nextNode
    return False
```

12)

Por propiedad, un grafo de n vértices tiene n-1 aristas en total. Si tenemos un árbol normal de m vértices, entonces este tendrá m - 1 aristas. Si le agregamos una arista más, obtenemos m aristas, por lo que no se cumple la propiedad y es falso.



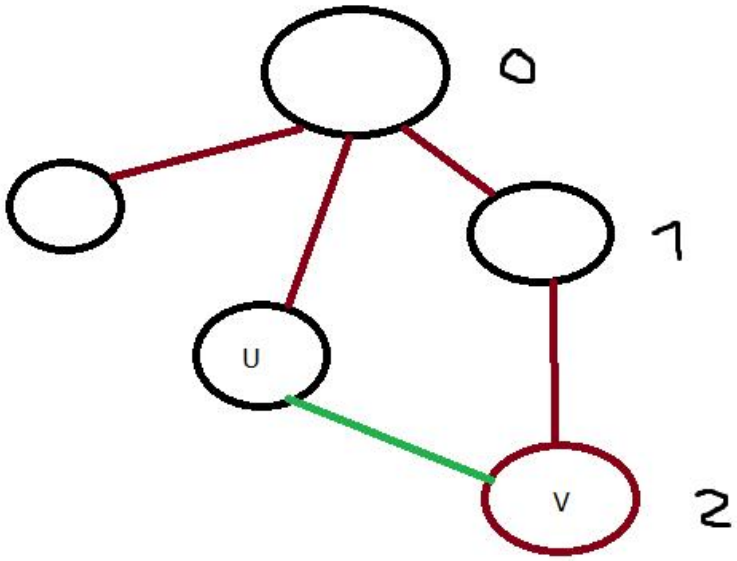
8 Vértices y 7 aristas. No hay ciclos



8 Vértices y 8 aristas. Se forma un ciclo

13)

Si una arista está conectada tanto a v como a u, eso significa que v está un nivel anterior o superior al de u. El algoritmo BFS recorre el grafo por niveles, añadiendo las aristas al árbol teniendo en cuenta eso. Si una arista (v,u) no ha sido agregada, significa que el vértice v fue encontrado añadiendo los vértices adyacentes a otro vértice del nivel de u y que se encuentra en el nivel inferior a este. Puesto que si no lo estuviese, hubiese sido agregado al recorrer u, y la arista (v, u) si existiría en el árbol BFS.



Por ejemplo, en este árbol, la arista (u,v) (La verde), no fue agregada porque el BFS recorrió añadiendo los vértices por niveles, y en el nivel 1 recorrió desde los vértices de la derecha hacia la izquierda y se encontró una arista que une a v. Por lo explicado anteriormente, el nivel del vértice u y v, difiere en 1 a lo sumo, puesto que de lo contrario, hubiese sido agregada al agregar las aristas de u, ya que va recorriendo por niveles.

14)
PRIM(Grafo)

```
def PRIM(Graph):
    Q = []
    n = len(Graph)
    newGraph = []
    visited = [0]
    Q = neighbourEdges(Graph, 0, Q, visited)
    edge = 0
    while (Q != []) and (len(newGraph) != (n-1)):
        edge = getMinor(Q, visited)

        if edge != None:
            newGraph.append(edge)
        else:
            return newGraph
        visited.append(edge[1])
        Q = neighbourEdges(Graph, edge[1], Q, visited)
    return newGraph

def getMinor(Q, visited):
    aux = None
    cont = 0
    while cont != len(Q):
        if (Q[cont][1] in visited):
            Q.pop(cont)
        cont += 1
    for i in range(len(Q)):
        if aux == None:
                aux = i
        else:
            if Q[aux][2] > Q[i][2]:
                aux = i
    if aux == None:
        return None
    aux2 = Q[aux]
    Q.pop(aux)
    return aux2

def neighbourEdges(Graph, v, Q, visited):
    for i in range(0, len(Graph)):
        if i not in visited:
            if Graph[v][i][0] != 0:
                Q.append((v, i, Graph[v][i][1]))
    return Q
```

15)
KRUSKAL(Grafo)

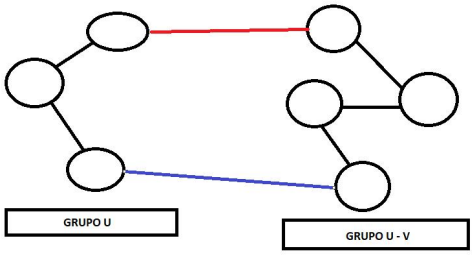
```
def KRUSKAL(Graph):
    edges = getEdges(Graph)
    edges.sort()
    newGraph = []
    parent, contador = makeSet(len(Graph))
    for i in edges:
        if union(i[1], i[2], parent, contador) == True:
            newGraph.append((i[1], i[2], i[0]))
    return newGraph

def getEdges(Graph):
    edges = []
    for i in range(0, len(Graph)):
        for j in range(i, len(Graph)):
            if Graph[i][j][0] != 0:
                edges.append((Graph[i][j][1], i, j))
    return edges

"""
Se usa para verificar los ciclos en KRUSKAL, crea varios conjuntos (al principio con cantidad de elementos igual a vértices) y
los va uniendo en KRUSKAL.
Si en un momento esos conjuntos están unidos, retorna false. Caso contrario, los une y retorna True
contador es aproximadamente la altura de cada subárbol y se usa para más o menos intentar balancear el árbol que va quedando.
"""
def makeSet(n):
    parent = []
    contador = []
    for i in range(n):
        parent.append(i)
        contador.append(0)
    return parent, contador

def find(x, parent):
    if parent[x] != x:
        parent[x] = find(parent[x], parent)
    return parent[x]

def union(x, y, parent, contador):
    raizX = find(x, parent)
    raizY = find(y, parent)
    if raizX == raizY:
        return False

    if contador[raizX] < contador[raizY]:
        parent[raizX] = raizY
    elif contador[raizX] > contador[raizY]:
        parent[raizY] = raizX
    else:
        parent[raizY] = raizX
        contador[raizX] = contador[raizX] + 1
    return True
```

16)

Si tenemos dos árboles abarcadores mínimos U y V-U y existe al menos una arista (u, v) tal que u pertenece a U y v pertenece a V-U, entonces, sí o sí pertenecerá al árbol abarcador mínimo (siempre que sea de coste mínimo). Esto se puede afirmar gracias a que la arista es de costo mínimo, puesto que un AACM como su nombre lo índica, posee todas las aristas de menor valor, y, al existir por lo menos la arista (u, v), entonces el árbol U y el árbol V-U deberán conectarse necesariamente, de lo contrario quedaría un bosque.

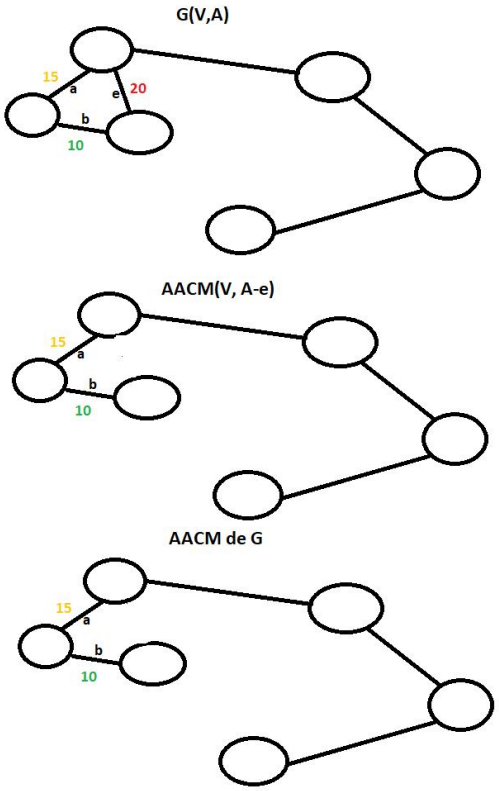
Si hubiesen dos aristas que conectasen U y V-U, llámese (u1, v1), igualmente (u, v) sería la que prevalecería, puesto que, nuevamente, es la arista con menor coste entre las dos. Si sucediese que ambas tuvieran el mismo valor, pues habrían dos árboles abarcadores posibles, y seguiría siendo verdadero, ya que se puede seleccionar el que tiene a (u,v).

En el ejemplo se muestra como se seleccionaría la roja en el caso de que tuviera menor valor que la azul, y si ambas tuviesen el mismo, pues se podría elegir nuevamente el de la roja para formar el AACM



17)

Sabemos que en un AACM no pueden existir ciclos y siempre se seleccionan las aristas de menor costo. Por ello, si se quita la arista que provoca el ciclo y además es la de mayor coste, entonces se generará un grafo sin ese ciclo, en el que si se hace el AACM no habrá problema al agregar aristas, ya que no se producirá un ciclo. Ese AACM será igual que el AACM del grafo original, ya que, al ser un árbol de costo mínimo, siempre se seleccionarán las aristas de menor coste, provocando que la arista eliminada (e), no se encuentre igualmente.



18)
Esto es demostrado por la propiedad de Kruskal, si G(V, E) es un grafo conexo, no dirigido y ponderado.
Si A es un subconjunto de E que está incluido en algún AACM
Si C = (Vc, Ec) es una componente conexa en el bosque Ga = (V, A).
Si (u, v) es una arista de costo mínimo que conecta C con otro componente de Ga, entonces existe un AACM para G que contiene (u, v) entre sus aristas.
Por ello, el enunciado es verdadero.
19)
1-En vez de seleccionar y guardar las aristas de menor coste, guardar las de mayor coste
2-Seleccionar las aristas en un orden arbitrario, sin tener en cuenta el coste.
3-Se podría intentar cambiar los valores de las aristas pertenecientes a E con el menor coste posible. De esta manera, existe al menos un AACM que cumpla con lo solicitado
20)
Si se tiene en cuenta que toda arista tiene coste mínimo, entonces es posible usar un recorrido para encontrar las aristas a agregar. Ya que es posible agregar cualquier arista siempre y cuando no se forme un ciclo

```
Matriz =[len(Graph), len(Graph)]
Bfs(Graph, 0):
for i in Graph
v = i
  for i in ady[v]
    If i == "W"
      M[v, i] = 1
      M[i, v] = 1
```

21)
shortestPath(Grafo, s, v) (o dijkstra)

```python
class dijVertex:
    value = None
    d = math.inf
    pi = None

def shortestPath(Grafo, s, v):
    return dijkstra(Grafo, s, v)

def initRelax(G, vertices, s):
    for i in range(len(G)):
        v = dijVertex()
        v.value = i
        vertices.append(v)
    vertices[s].d = 0
    vertices[s].pi = s
    return vertices

def relax(Graph, Q, vertices, u, v):
    if vertices[v].d > vertices[u].d + Graph[u][v][1]:
        vertices[v].d = vertices[u].d + Graph[u][v][1]
        aux = vertices[u].d + Graph[u][v][1]
        vertices[v].pi = u
        for i in range(len(Q)):
            if Q[i][0] == v:
                aux2 = i
        Q.pop(aux2)
        if len(Q) == 0:
            Q.insert(0, (v, vertices[u].d + Graph[u][v][1]))
        else:
            for i in range(len(Q)):
                if aux <= Q[i][1]:
                    Q.insert(i, (v, vertices[u].d + Graph[u][v][1]))
                    break
    return vertices, Q
```

```python
def dijkstra(Graph, s, v):
    vertices = []
    vertices = initRelax(Graph, vertices, s)
    S = []
    Q = []
    Q.append((s, 0))
    for i in range(1, len(Graph)):
        if i != s:
            Q.append((i, math.inf))
    while len(Q) > 0:
        u = Q.pop(0)
        S.append(u)
        for i in range(0, len(Graph)):
            if Graph[u[0]][i][0] == 1:
                if i not in S:
                    vertices, Q = relax(Graph, Q, vertices, u[0], i)
    aux = None
    definitiveList = []
    if vertices[v].pi != None:
        definitiveList.append(v)
    while aux != s:
        aux = vertices[v].pi
        if aux == None:
            break
        v = vertices[v].pi
        definitiveList.insert(0, aux)
    return definitiveList
```