A partir de la siguiente definición:

```
Graph = Array(n,LinkedList())
```

Donde **Graph** es una representación de un grafo **simple** mediante listas de adyacencia resolver los siguiente ejercicios

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

```
def createGraph(vertices, edges):
    n = length(vertices)
    if n <= 0:
        return
    graph = Array(n,LinkedList())
    node = vertices.head
    while node != None:
        nodo = Node()
        nodo.value = node.value
        index = hash(node.value, n)
        graph[index] = LinkedList()
        graph[index].head = nodo
        node = node.nextNode

    node = edges.head
    while node != None:
        addV(node.value[0], node.value[1], n, graph)
        addV(node.value[1], node.value[0], n, graph)
        node = node.nextNode

    return graph
```

Ejercicio 2

def existP
Descr busca si exist
Entra cencia, **v1** y **v2**
Salid **False** en caso

```
def existPath(graph, v1, v2):
    bfs(graph, v1)
    n = len(graph)
    vertex = graph[hash(v2, n)].head
    for i in range(n):
        if vertex.value[0] == v1:
            return True
        else:
            vertex = graph[hash(vertex.value[2], n)].head
    return False



def bfs(graph, s):
    for i in graph:
        value = i.head.value
        i.head.value = [value, "white",None,0]

    n = len(graph)
    vertex = graph[hash(s,n)].head
    vertex.value[1] = "grey"
    vertex.value[3] = 0
```

Ejercicio 3

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.

```
def isConnected(graph):
    a = graph[0].head.value
    bfs(graph, a)
    for i in graph:
        if i.head.value[1] == "white":
            return False
    return True
```

Ejercicio 4

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.

```
def isTree(graph):
    return isConnected(graph) and cycleVerify(graph)


def cycleVerify(graph):
    s = graph[0].head.value
    for i in graph:
        value = i.head.value
```

Ejercicio 5

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.

```python
def isComplete(graph):
    n = len(graph)
    j = 0
    for i in graph:
        node = i.head
        letter = i.head.value
        while node != None:
            j += 1
            node = node.nextNode
            if node != None:
                if node.value == letter:
                    return False
        if j == n:
            j = 0
        else:
            return False
    return True
```

Ejercicio 6

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.

```
def convertTree(graph):
    s = graph[0].head.value
    for i in graph:
        value = i.head.value
        i.head.value = [value, "white",None]

    n = len(graph)
    vertex = graph[hash(s,n)].head
    vertex.value[1] = "grey"
    Q = LinkedList()
    edges = LinkedList()
    enqueue(Q,vertex)
    while Q.head != None:
        u = dequeue(Q)
        letter = u.value[0]
        colour = u.value[1]
        if u.nextNode != None:
            node = u.nextNode
            while node != None:
                if graph[hash(node.value,n)].head.value[1] == "grey":
                    nodo = Node()
                    nodo.value = [letter,node.value]
                    add(edges, nodo)
                node = node.nextNode
        while u != None:

    def dfs(graph):
        for i in graph:
            value = i.head.value
            i.head.value = [value,"white",None,0,0]

        n = len(graph)
        time = 0
        for i in graph:
            if i.head.value[1] == "white":
                dfsR(graph, i.head, n, time)

    def dfsR(graph,u,n, time):
        time += 1
        u.value[3] = time
        u.value[1] = "gray"
        letter = u.value[0]
        while u != None:
            vertex = graph[hash(u.value[0],n)].head
            if vertex.value[1] == "white":
                vertex.value[2] = letter
                dfsR(graph, vertex, n, time)
            u = u.nextNode

        if letter != None:
            g = graph[hash(letter,n)].head
            g.value[1] = "black"
            time += 1
            g.value[4] = time
```

```
def countConnections(graph):
    conexNumber = 0
    for i in graph:
        value = i.head.value
        i.head.value = [value,"white",None,0,0]

    n = len(graph)
    time = 0
    for i in graph:
        if i.head.value[1] == "white":
            conexNumber += 1
            dfsR(graph, i.head, n, time)

    return conexNumber
```

Ejercicio 8

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.

```
def convertToBFSTree(graph, v):
    for i in graph:
        value = i.head.value
        i.head.value = [value, "white",None]

    n = len(graph)
    vertex = graph[hash(v,n)].head
    vertex.value[1] = "grey"
    Q = LinkedList()
    enqueue(Q,vertex)
    j = 0
    graphReturn = Array(n, LinkedList())
    while Q.head != None:
        u = dequeue(Q)
        letter = u.value[0]
        colour = u.value[1]
        graphReturn[j] = LinkedList()
        node = Node()
        node.value = letter
        graphReturn[j].head = node
        node = graphReturn[j].head
        while u != None:
            if colour == "white":
                vertex = graph[hash(u.value,n)].head
                lyric = vertex.value[0]
                vertex.value[1] = "grey"
                vertex.value[2] = letter
                nodoLyric = Node()
                nodoLyric.value = lyric
                node.nextNode = nodoLyric
                node = node.nextNode
                enqueue(Q, vertex)
            u = u.nextNode
            if u != None:
                colour = graph[hash(u.value, n)].head.value[1]
        j += 1
```

Ejercicio 9

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.

```
def convertToDFSTree(graph, v):
    for i in graph:
        value = i.head.value
        i.head.value = [value,"white",0]

    n = len(graph)
    vertex = graph[hash(v, n)].head
    colour = vertex.value[1]
    i = vertex
    j = 0
    while vertex != None:
        if colour == "white":
            convertToDFSTreeR(graph, i, n,j)
        vertex = vertex.nextNode
        if vertex != None:
            i = graph[hash(vertex.value,n)].head
            colour = i.value[1]

    graphR = Array(n, LinkedList())
    for i in graph:
        nodo = i.head
        position = nodo.value[2]
        graphR[position] = LinkedList()
        graphR[position].head = nodo

    return graphR

def convertToDFSTreeR(graph,u,n,j):
    u.value[1] = "gray"
    u.value[2] = j
    while u != None:
        vertex = graph[hash(u.value[0],n)].head
        if vertex.value[1] == "white":
            j += 1
            sentinel = convertToDFSTreeR(graph, vertex, n, j)
            if sentinel > j:
                j = sentinel
        u = u.nextNode
    return j
```

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.

```
def bestRoad(graph, v1, v2):
    bfs(graph,v1)
    n = len(graph)
    edges = LinkedList()
    vertex = graph[hash(v2,n)].head
    letter = vertex.value[2]
    while True:
        add(edges, (letter,vertex.value[0]))
        vertex = graph[hash(letter, n)].head
        if letter == v1:
            return edges
        letter = vertex.value[2]
```

Ejercicio 11 (Opcional)

Implementar la función que responde a la siguiente especificación.

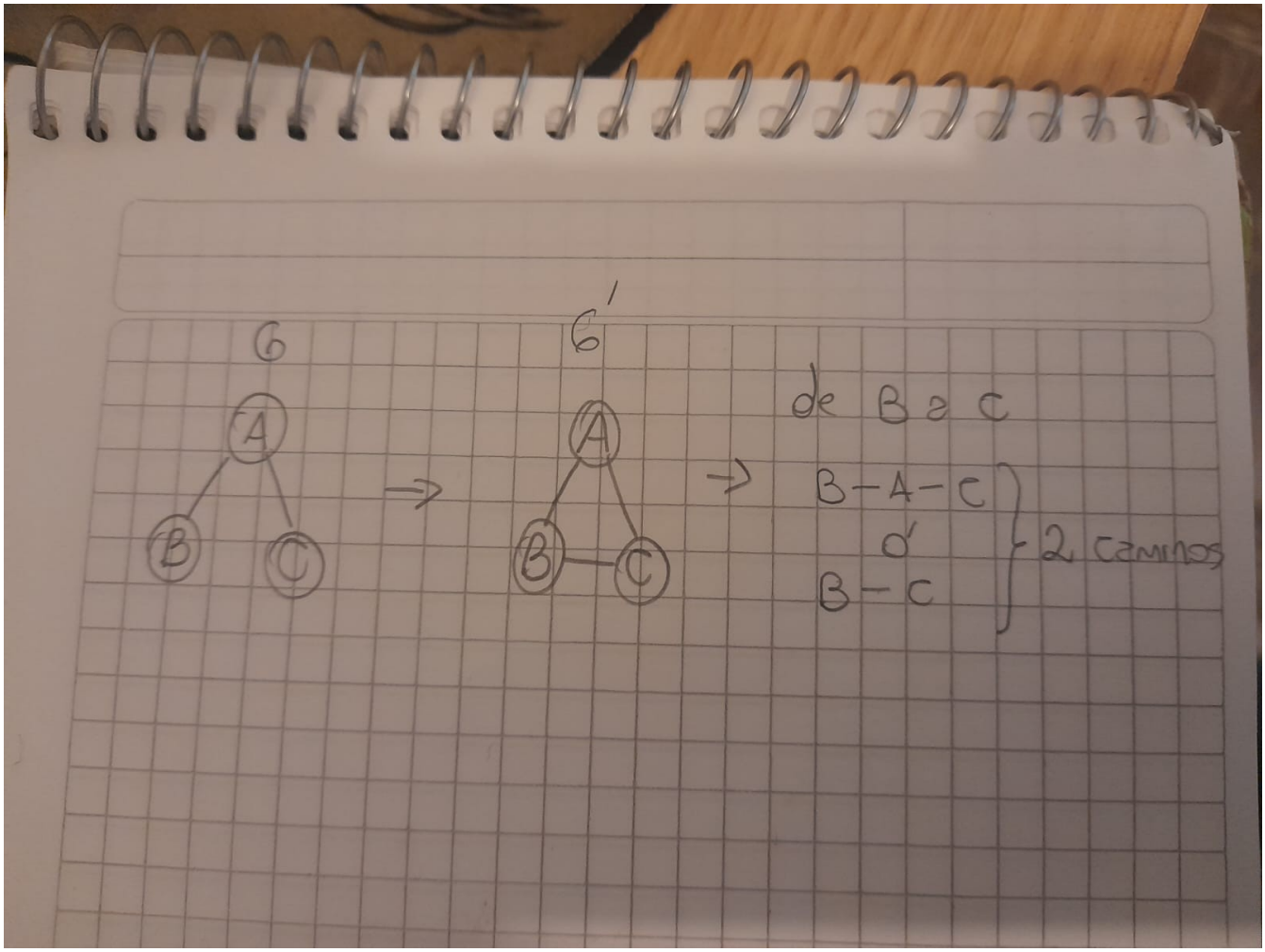
def isBipartite(Grafo):
Descripción: Implementa la operación es bipartito
Entrada: Grafo con la representación de Lista de Adyacencia.
Salida: retorna **True** si el grafo es bipartito.

NOTA: Un grafo es **bipartito** si no tiene ciclos de longitud impar.

Ejercicio 12

Demuestre que si el grafo G es un árbol y se le agrega una arista nueva entre cualquier par de vértices se forma exactamente un ciclo y deja de ser un árbol.

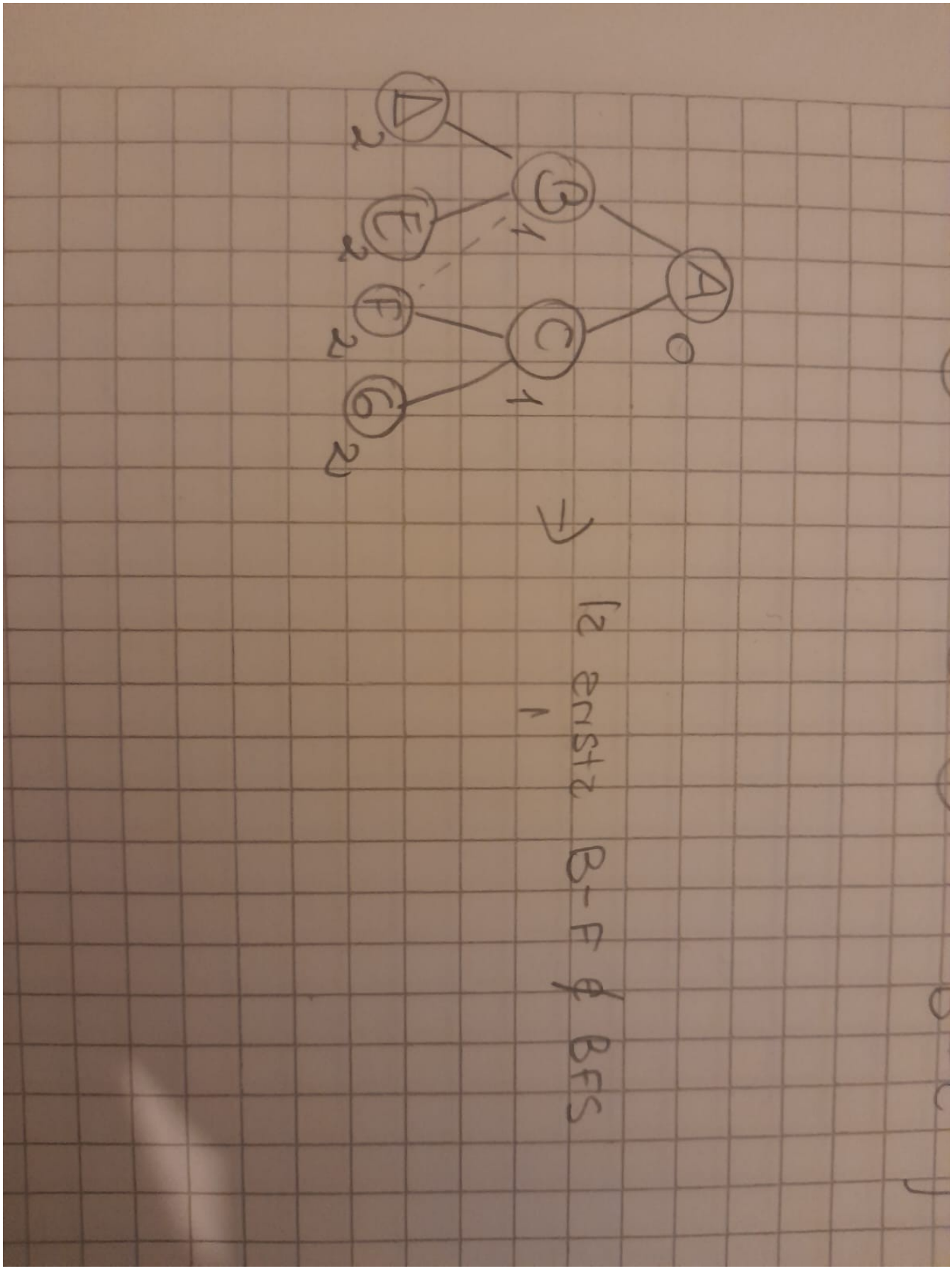
G es un árbol, sabemos que es un grafo conexo y acíclico.
Ahora, supongamos que agregamos una arista nueva entre dos vértices distintos en el grafo G.
Llamemos a estos vértices B y C.
Al agregar la arista entre B y C, se crea una conexión directa entre ellos, lo que implica que hay al menos un camino entre B y C.
Sin embargo, dado que G es un árbol, solo puede haber un único camino entre cualquier par de vértices. Si agregamos una arista entre B y C, habrá dos caminos posibles para ir de B a C: uno a través de la nueva arista y otro a través del camino existente en el árbol.
La existencia de dos caminos entre B y C implica que se ha formado un ciclo en el grafo resultante.
Podemos seguir el camino desde B a C a través de la nueva arista y luego volver de C a B a través del camino existente en el árbol, formando así un ciclo cerrado.
Por lo tanto, el grafo resultante ya no es acíclico, lo que contradice la definición de un árbol.



Ejercicio 13

Demuestre que si la arista (u,v) no pertenece al árbol BFS, entonces los niveles de u y v difieren a lo sumo en 1.

Como (u, v) no pertenece al árbol BFS, la única forma en que u puede estar a una distancia mayor que v+1 desde el vértice raíz es si existe una arista más corta que conecta u directamente con un vértice que se encuentra más cerca del vértice raíz que v.



Si observamos los niveles de los vértices u (B) y v (F), podemos ver que difieren en 1. El nivel de u es 1 y el nivel de v es 2. No puede haber una diferencia mayor a 1 en los niveles, ya que en un árbol BFS cada vértice se encuentra a la distancia más corta desde el vértice raíz.

Si supusiéramos que los niveles de u y v difieren en más de 1, por ejemplo, el nivel de u es mayor que el nivel de v más 1, tendríamos una contradicción.

Ejercicio 14

Implementar la función que responde a la siguiente especificación.

```
def PRIM(Grafo):
     Descripción: Implementa el algoritmo de PRIM
     Entrada: Grafo con la representación de Matriz de Adyacencia.
     Salida: retorna el árbol abarcador de costo mínimo
```

```
def prim(graph):
    n = len(graph)
    v = graph[0].head
    w = graph[0].head.nextNode
    visited = LinkedList()
    add(visited, v.value)
    Q = LinkedList()
    graphR = Array(n, LinkedList())
    j = 0
    for i in graph:
        graphR[j] = LinkedList()
        node = Node()
        node.value = i.head.value
        graphR[j].head = node
        j += 1

    while w != None:
        enqueue(Q, [(v.value,w.value),random.randint(1,9)])
        w = w.nextNode

    u = dequeue(Q)
    node = Node()
    node.value = (u[0][1],u[1])
    graphR[0].head.nextNode = node
    while u != None:
        w = u
        add(visited, w[0][1])
        r = graphR[hash(w[0][0],n)]
        if r.head.nextNode == None:
            node = Node()
            node.value = (w[0][1], w[1])
            r.head.nextNode = node
        elif r.head.nextNode.value[1] > u[1]:
            node = Node()
            node.value = (w[0][1],w[1])
            r.head.nextNode = node
        v = graph[hash(w[0][1],n)].head
        w = graph[hash(w[0][1],n)].head.nextNode
        while w != None:
            if search(visited, w.value) == None:
                enqueue(Q, [(v.value,w.value), random.randint(1,9)])
            w = w.nextNode
        u = dequeue(Q)
    return graphR
```

Ejercicio 15

def KRUSKAL(Grafo):
Descripción: Implementa el algoritmo de KRUSKAL

Entrada: Grafo con la representación de Matriz de Adyacencia.
Salida: retorna el árbol abarcador de costo mínimo

```
def kruskal(graph):
    n = len(graph)
    sets = Array(n,("",""))
    j = 0
    A = LinkedList()
    B = LinkedList()
    for i in  graph:
        add(A,(i.head.value))
        sets[j] = (i.head.value, i.head.value)
        j += 1
    edges = LinkedList()
    sortEdges(graph, edges)
    nodo = edges.head
    while nodo != None:
        if find(sets,nodo.value[0],n) != find(sets, nodo.value[1],n):
            add(B,((nodo.value[0], nodo.value[1])))
            union(sets, nodo.value[0], nodo.value[1], n)
        nodo = nodo.nextNode
    return createGraph(A,B)

def sortEdges(graph, edges):
    visited = LinkedList()
    for i in graph:
        nodo = i.head
        vertex = nodo.value
        nodo = nodo.nextNode
        while nodo != None:
            if search(visited, nodo.value[0]) == None:
                enqueuePriority(edges, (vertex, nodo.value[0], nodo.value[1]))
            nodo = nodo.nextNode
        add(visited, vertex)

def union(sets, u, v, n):
    newGroup = find(sets, u, n)
    otherGroup = find(sets, v, n)
    sets[hash(otherGroup,n)] = (otherGroup,newGroup)

def find(sets,vertex,n):
    if sets[hash(vertex, n)][1] == vertex:
        return vertex

    group = find(sets, sets[hash(vertex,n)][1], n)
    sets[hash(vertex,n)] = (vertex,group)
    return group
```

```
def enqueuePriority(Q, value):
    node = Node()
    node.value = value
    head = Q.head
    flag = True
    if head == None:
        Q.head = node
    else:
        pre = head
        while head != None:
            if head.value[2] > node.value[2]:
                if head.value == Q.head.value:
                    node.nextNode = Q.head
                    Q.head = node
                else:
                    nodo = pre.nextNode
                    pre.nextNode = node
                    node.nextNode = nodo
                flag = False
                break
            pre = head
            head = head.nextNode
        if flag:
            pre.nextNode = node
```

Ejercicio 16

Demostrar que si la arista (u,v) de costo mínimo tiene un nodo en U y otro en V - U, entonces la arista (u,v) pertenece a un árbol abarcador de costo mínimo.

Parte 4

Ejercicio 17

Sea **e** la arista de mayor costo de algún ciclo de **G(V,A)** . Demuestre que existe un árbol abarcador de costo mínimo **AACM(V,A-e)** que también lo es de **G.**

Ejercicio 18

Demuestre que si unimos dos **AACM** por un arco (arista) de costo mínimo el resultado es un nuevo **AACM**. (Base del funcionamiento del algoritmo de **Kruskal**)

Ejercicio 19

Explique qué modificaciones habría que hacer en el algoritmo de Prim sobre el grafo no dirigido y conexo **G(V,A)**, o sobre la función de costo **c(v1,v2)-> R** para lograr:

1. Obtener un árbol de recubrimiento de costo máximo.
2. Obtener un árbol de recubrimiento cualquiera.

3. Dado un conjunto de aristas E $\in$ A, que no forman un ciclo, encontrar el árbol de recubrimiento mínimo $G^c(V,A^C)$ tal que E $\in$ A^c.

1. Obtener un árbol de recubrimiento de costo máximo.
Para eso en vez de iterar y buscar la arista de costo minimo, busco la arista de costo máximo

2. Obtener un árbol de recubrimiento cualquiera
No buscaria la arista de costo maximo ni minimo, si no mas bien agarraria sus valores y aplicaria un ramdon e eligiria 1

3. Dado un conjunto de aristas E $\in$ A, que no forman un ciclo, encontrar el árbol de recubrimiento mínimo Gc(V,AC) tal que E $\in$ Ac.
Inicializar las distancias de los vértices en E a 0 y las distancias de los vértices no en E a un valor muy grande o infinito. Considerar las aristas de E primero Actualizar las distancias solo si el peso de la arista candidata es menor

Ejercicio 20

Sea **G<V, A>** un grafo conexo, no dirigido y ponderado, donde todas las aristas tienen el mismo costo. Suponiendo que G está implementado usando matriz de adyacencia, haga en pseudocódigo un algoritmo **$O(V^2)$** que devuelva una matriz M de VxV donde: M[u, v] = 1 si (u,v) $\in$ A y (u, v) estará obligatoriamente en todo árbol abarcador de costo mínimo de **G**, y cero en caso contrario.

```
Algoritmo matriz_AACM(G)
    n = número de vértices en G
    M = matriz de tamaño n x n inicializada con ceros
    Para cada vértice u de 0 a n-1:
        Para cada vértice v de u+1 a n-1:
        M[u, v] = 1
    Devolver M
```

Parte 5

Ejercicio 21

Implementar el Algoritmo de Dijkstra que responde a la siguiente especificación

```
def shortestPath(Grafo, s, v):
```

Descripción: Implementa el algoritmo de Dijkstra
Entrada: Grafo con la representación de Matriz de Adyacencia, vértice de inicio **s** y destino **v**.
Salida: retorna la lista de los vértices que conforman el camino iniciando por **s** y terminando en **v**. Devolver NONE en caso que no exista camino entre **s** y **v**.

```
def dijkstra(graph,s,v):
    n = len(graph)
    initRelax(graph, s)
    S = LinkedList()
    Q = LinkedList()
    enqueue(Q, graph[hash(s, n)].head)
    while Q != None:
        nodo = dequeue(Q)
        u = nodo
        add(S, u.value[0])
        nodo = nodo.nextNode
        while nodo != None:
            if search(S, nodo.value[0]) == None:
                v = graph[hash(nodo.value[0],n)]
                w = nodo.value[1]
                relax(u.value, v.head.value, w, Q, graph)
            nodo = nodo.nextNode

def minQueue(graph, Q):
    for i in graph:
        if i.head.value[1] != None:
            nodo = Node()
            nodo.value = i.head
            if Q.head == None:
                Q.head = nodo
                break

def initRelax(graph,s):
    n = len(graph)
    for i in graph:
        letter = i.head.value
        i.head.value = [letter, None, None]
    graph[hash(s,n)].head.value[1] = 0

def relax(u,v, w, Q, graph):
    if v[1] == None or v[1] > (u[1] + w):
        v[1] = u[1] + w
        v[2] = u[0]
        enqueue(Q, graph[hash(v[0],5)].head)
```

Ejercicio 22 (Opcional)

Sea **G = <V, A>** un grafo dirigido y ponderado con la función de costos C: A → R de forma tal

que C(v, w) > 0 para todo arco <v, w> ∈ A. Se define el costo C(p) de todo camino p = <v0, v1, …, vk> como C(v0, v1) * C(v1, v2) * … * C(vk - 1, vk).

a) Demuestre que si p = <v0, v1, …, vk> es el camino de menor costo con respecto a C en ir de v0 hacia vk, entonces <vi, vi + 1, .., vj> es el camino de menor costo (también con respecto a C) en ir de vi a vj para todo 0 ≤ i < j ≤ k.
b) ¿Bajo qué condición o condiciones se puede afirmar que con respecto a C existe camino de costo mínimo entre dos vértices a, b ∈V? Justifique su respuesta.
c) Demuestre que, usando la función de costos C tal y como la dan, no se puede aplicar el algoritmo de Dijkstra para hallar los costos de los caminos de costo mínimo desde un vértice de origen s hacia el resto.
d) Plantee un algoritmo, lo más eficiente en tiempo que usted pueda, que determine los costos de los caminos de costo mínimo desde un vértice de origen s hacia el resto usando la función de costos C.
e) Suponiendo que C(v, w) > 1 para todo <v, w> ∈A, proponga una función de costos C':A → R y además la forma de calcular el costo C'(p) de todo camino p = <v0, v1, …, vk> de forma tal que: aplicando el algoritmo de Dijkstra usando C', se puedan obtener los costos (con respecto a la función original C) de los caminos de costo mínimo desde un vértice de origen s hacia el resto. Justifique su respuesta.

A tener en cuenta:

1. **Usen lápiz y papel primero**
2. ~~**No se puede utilizar otra Biblioteca más allá de algo1.py y las bibliotecas desarrolladas durante Algoritmos y Estructuras de Datos I.**~~