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
```

```python
def createGraph(listV,listA):
    #Verificar que la lista de vértices no esté vacía.
    if listV == []:
        return "Incorrecto, el grafo no contiene vértices."
    else:
        #Definir la lista de adyacencia del grafo.
        listAdjacency = []
        #Recorrer los vértices del grafo.
        for i in listV:
            #Definir el diccionario que irá dentro de la lista de adyacencia del grafo.
            dictionary = {}
            #Definir la lista de adyacencia de cada vértice del grafo.
            listAdjacencyInDictionary = []
            #Recorrer la lista de aristas del grafo.
            if listA != []:
                for j in listA:
                    #Validar el caso de ingresar aristas repetidas(con vértices ordenados o no).
                    if i == j[0]:
                        if dictionary == {}:
                            listAdjacencyInDictionary.append(j[1])
                            dictionary[i] = listAdjacencyInDictionary
                        else:
                            if j[1] not in dictionary[i]:
                                listAdjacencyInDictionary.append(j[1])
                                dictionary[i] = listAdjacencyInDictionary
                    elif i == j[1]:
                        if dictionary == {}:
                            listAdjacencyInDictionary.append(j[0])
                            dictionary[i] = listAdjacencyInDictionary
                        else:
                            if j[0] not in dictionary[i]:
                                listAdjacencyInDictionary.append(j[0])
                                dictionary[i] = listAdjacencyInDictionary
            #Si el diccionario o la listA estan vacíos es porque el vértice no tiene vértices adyacentes.
            if dictionary == {}:
                dictionary[i] = None
                listAdjacency.append(dictionary)
            else:
                listAdjacency.append(dictionary)
    return listAdjacency
```

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.

```python
def existPath(Graph,v1,v2):
    #Verificar que el grafo no se encuentre vacío.
    if Graph == []:
        return "El grafo está vacío."
    #Verificar que los vértices v1 y v2 pertenezcan al grafo.
    countV1 = 0
    countV2 = 0
    for i in range(len(Graph)):
        if v1 not in Graph[i]:
            countV1 += 1
        if v2 not in Graph[i]:
            countV2 += 1
    if countV1 == len(Graph) and countV2 == len(Graph):
        return f"Los vértices {v1} y {v2} no pertenece al grafo."
    elif countV1 == len(Graph):
        return f"El vértice {v1} no pertenece al grafo."
    elif countV2 == len(Graph):
        return f"El vértice {v2} no pertenece al grafo."
    else:
        #Obtener la lista de adyacencia de v1 respecto del DFS.
        DFSv1 = convertToDFSTree(Graph,v1)
        #Verificar que v1 no sea un vértice no conexo.
        if DFSv1 == "Vértice no conexo":
            return False
        else:
            #Verificar si existe un camino entre v1 y v2.
            for i in range(len(DFSv1)):
                if v2 in DFSv1[i]:
                    return True
            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.

```python
def isConnected(Graph):
    #Verificar que el grafo no esté vecío.
    if Graph == []:
        return "El grafo se encuentra vacío."
    else:
        #Si el grafo no esta vacío entonces calcular su DFS.
        lista = []
        lista.append(list(Graph[0].keys()))
        GraphDFS = convertToDFSTree(Graph,lista[0][0])
        if len(GraphDFS) == len(Graph):
            return True
        else:
            return False
#Grafo ponderado conexo.
def isConnectedWeighted(Graph):
    #Verificar que el grafo no esté vecío.
    if Graph == []:
        return "El grafo se encuentra vacío."
    else:
        #Si el grafo no esta vacío entonces calcular su DFS.
        lista = []
        lista.append(list(Graph[0].keys()))
        GraphDFS = convertWeightedToDFSTree(Graph,lista[0][0])
        if len(GraphDFS) == len(Graph[0]):
            return True
        else:
            return False
```

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.

```python
def isTree(Graph):
    #Verificar que el grafo no esté vacío.
    if Graph == []:
        return "El Grafo se encuentra vacio."
    #Obtener un vértice raíz del grafo.
    List = []
    ListNodeVertex = []
    for i in range(len(Graph)):
        List.append(list(Graph[i].keys()))
        #Obtener la lista de vértices de tipo nodeVertex.
        ListNode = insertVertex(List[i][0],ListNodeVertex)
    vertex = List[0][0]
    #Crear una lista para los vértices visitados.
    listVisited = []
    listVisited.append(vertex)
    #Crear una cola para colocar los vértices adyacentes no visitados.
    Queue = []
    Queue.append(vertex)
    while Queue != []:
        u = Queue.pop(0)
        #Verificar que el vértice se encuentre en el grafo.
        for i in range(len(Graph)):
            if u in Graph[i]:
                adjacencyNodes = Graph[i][u]
                break
            else:
                adjacencyNodes = None
        #Verificar que el vértice u sea conexo.
        if adjacencyNodes is not None:
            #Agregar el padre de cada vértice.
            for index in ListNode:
                if index.value in adjacencyNodes:
                    if index.parent is None:
                        index.parent = u
            #Buscar el nodo vértice para evaluar el ciclo.
            for i in ListNode:
                if i.value == u:
                    nodeV = i
            #Recorrer la lista de adyacencia del vértice anterior(u).
            for i in adjacencyNodes:
            #Verificar que el vértice adyacente de u no se encuentre en la lista de visitados.
                if i not in listVisited:
                    Queue.append(i)
                    listVisited.append((i))
                else:
                    if nodeV.parent != i:
                        return False
    return True
```

```python
def insertVertex(v,ListNode):
    vertexNode = nodeVertexDFS()
    vertexNode.value = v
    ListNode.append(vertexNode)
    return ListNode
```

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.

```python
def isComplete(Graph):
    #Verificar que el grafo no esté vacío.
    if Graph == []:
        return "El Grafo se encuentra vacio."
    #Verificar que el grafo sea conexo.
    connectedGraph = isConnected(Graph)
    if connectedGraph is False:
        return False
    #Guardar en una lista los vértices del grafo.
    List = []
    for i in Graph:
        List.append(list(i.keys()))
    #Verificar que existe una arista para todo par de vértices.
    for i in range(len(Graph)):
        #Key del diccionario donde está el vértice.
        index = List[i][0]
        for vertex in List:
            if vertex[0] != index:
                if vertex[0] not in Graph[i][index]:
                    return False
    return True
```

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.

```python
def convertTree(Graph):
    #Verificar que el grafo no esté vacío.
    if Graph == []:
        return "El Grafo se encuentra vacio."
    #Verificar si el grafo ya es un árbol.
    graphTree = isTree(Graph)
    if graphTree is True:
        return "El grafo ya es un árbol."
    #Obtener un vértice raíz del grafo.
    List = []
    ListNodeVertex = []
    for i in range(len(Graph)):
        List.append(list(Graph[i].keys()))
        #Obtener la lista de vértices de tipo nodeVertex.
        ListNode = insertVertex(List[i][0],ListNodeVertex)
    vertex = List[0][0]
    #Crear una lista para los vértices visitados.
    listVisited = []
    listVisited.append(vertex)
    #Crear una cola para colocar los vértices adyacentes no visitados.
    Queue = []
    Queue.append(vertex)
    #Definir la lista donde estarán las aristas que producen ciclos.
    deleteEdges = []
    while Queue != []:
        #Desencolar el primer elemento de la cola.
        u = Queue.pop(0)
        #Verificar que el vértice se encuentre en el grafo.
        for i in range(len(Graph)):
            if u in Graph[i]:
                adjacencyNodes = Graph[i][u]
                break
            else:
                adjacencyNodes = None
        #Verificar que el vértice u sea conexo.
        if adjacencyNodes is not None:
            #Agregar el padre de cada vértice.
            for index in ListNode:
                if index.value in adjacencyNodes:
```


```python
            #Agregar el padre de cada vértice.
            for index in ListNode:
                if index.value in adjacencyNodes:
```


```python
                    if index.parent is None:
                        index.parent = u
            #Buscar el nodo vértice para evaluar el ciclo.
            for i in ListNode:
                if i.value == u:
                    nodeV = i
            #Recorrer la lista de adyacencia del vértice anterior(u).
            for i in adjacencyNodes:
            #Verificar que el vértice adyacente de u no se encuentre en la lista de visitados.
                if i not in listVisited:
                    Queue.append(i)
                    listVisited.append((i))
                else:
                    if nodeV.parent != i:
                        #Agrego a la lista las aristas que hacen que el grafo no sea un árbol.
                        deleteEdges.append((u,i))
    return deleteEdges
```

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.

```python
def countConnections(Graph):
    #Verificar que el grafo no esté vacío.
    if Graph == []:
        return "El grafo está vacío"
    #Obtener los vértices del grafo.
    listVertex = []
    for vertex in Graph:
        listVertex.append(list(vertex.keys()))
    #Definir los vértices del grafo como nodos con sus atributos.
    Lnodes = []
    for vertex in listVertex:
        listNodes = insertVertex(vertex[0], Lnodes)
    #Asignar el color a los nodos.
    for node in listNodes:
        node.color = "white"
        node.parent = None
    #Verificar la cantidad de componentes conexas.
    time = 0
    count = 0
    listDFS = []
    for node in listNodes:
        if node.color == "white":
            listDFSResult = countConnectionsRecursive(Graph,node,time,listNodes,listDFS)
            count += 1
    return listDFSResult, count
```

```python
def countConnectionsRecursive(Graph,node,time,listNodes,listDFS):
    if node.distance == 0:
        time += 1
        node.distance = time
        node.color = "gray"
    #Obtener la lista de adyacencia de cada vértice.
    for vertex in Graph:
        if node.value in vertex:
            adjacencyNodes = vertex[node.value]
    #Verificar que la lista de adyacencia de cada nodo no esté vacía.
    if adjacencyNodes != None:
        #Buscar los vértices adyacentes de tipo nodo.
        nodeList = []
        for vertex in adjacencyNodes:
            for nodeVertex in listNodes:
                if nodeVertex.value == vertex:
                    nodeList.append(nodeVertex)
        #Recorrer la lista de nodos y realizar el DFS.
        for nodeVertex in nodeList:
            if nodeVertex.color == "white":
                nodeVertex.parent = node
                #Agregar nodos a la lista final del DFS.
                dictionary = {}
                dictionary[node.value] = nodeVertex
                listDFS.append(dictionary)
                countConnectionsRecursive(Graph,nodeVertex,time,listNodes,listDFS)
    node.color = "black"
    time += 1
    node.time = time
    #Caso base (fin de la recursión)
    if node.parent == None:
        return listDFS
    else:
        countConnectionsRecursive(Graph,node.parent,time,listNodes,listDFS)
```

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.

```python
def convertToBFSTree(Graph, v):
    #Verificar que el grafo no este vacío.
    if Graph == []:
        return "El Grafo se encuentra vacio."
    #Verificar que el vértice raíz se encuentre en el grafo.
    count = 0
    for i in range(len(Graph)):
        if v not in Graph[i]:
            count += 1
    if count == len(Graph):
        return "El Vértice raíz no se encuentra en el grafo."
    #Crear una lista para los vértices visitados.
    listVisited = []
    listVisited.append(v)
    #Crear una cola para colocar los vértices adyacentes no visitados.
    Queue = []
    Queue.append(v)
    #Definir la lista de adyacencia del BFS.
    adjacencyList = []
    #Distancia desde la raíz a cualquier vértice.
    diccDistance = {}
    distance = 0
    diccDistance[v] = distance
    while Queue != []:
        #Definir el diccionario que irá en la lista de adyacencia del BFS.
        dictionary = {}
        #Lista de adyacencia que va dentro del diccionario.
        adjacencyListInDictionary = []
        #Desencolar el primer elemento de la cola.
        u = Queue.pop(0)
        #Verificar que el vértice se encuentre en el grafo.
        for i in range(len(Graph)):
            if u in Graph[i]:
                adjacencyNodes = Graph[i][u]
                break
            else:
                adjacencyNodes = None
        #Verificar que el vértice u sea conexo.
        if adjacencyNodes is not None:
            #Recorrer la lista de adyacencia del vértice anterior(u).
            for i in adjacencyNodes:
            #Verificar que el vértice adyacente de u no se encuentre en la lista de visitados.
                if i not in listVisited:
                    Queue.append(i)
                    listVisited.append((i))
                    diccDistance[i] = diccDistance[u] + 1
                    adjacencyListInDictionary.append((i,diccDistance[i]))
                    dictionary[u] = adjacencyListInDictionary
        #Agregar diccionarios vacíos a la lista del BFS (aquí se produce el ciclo).
        dictionary[u] = adjacencyListInDictionary
        adjacencyList.append(dictionary)
    return adjacencyList
```

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.

```python
def convertToDFSTree(Graph, v):
    #Verificar que el grafo no este vacio.
    if Graph == []:
        return "El Grafo se encuentra vacio."
    #Verificar que el vértice raíz se encuentre en el grafo.
    count = 0
    for i in range(len(Graph)):
        if v not in Graph[i]:
            count += 1
    if count == len(Graph):
        return "El Vértice raíz no se encuentra en el grafo."
    #Crear una lista para los vértices visitados.
    listVisited = []
    listVisited.append(v)
    #Crear una cola para colocar los vértices adyacentes no visitados.
    Queue = []
    Queue.append(v)
    #Definir la lista de adyacencia del DFS.
    adjacencyList = []
    adjacencyListResult = convertToDFSTreeRecursive(Graph,v,listVisited,Queue,adjacencyList)
    return adjacencyListResult
```

```python
def convertToDFSTreeRecursive(Graph,v,listVisited,Queue,adjacencyList):
    #Caso base de la recursividad.
    if Queue == []:
        return adjacencyList
    while Queue != None:
        #Definir el diccionario que irá en la lista de adyacencia del DFS.
        dictionary = {}
        #Lista de adyacencia que va dentro del diccionario.
        adjacencyListInDictionary = []
        #Si el grafo tiene un solo vértice.
        if len(Graph) == 1:
            dictionary[v] = adjacencyListInDictionary
            adjacencyList.append(dictionary)
            return adjacencyList
        #Verificar que el vértice se encuentre en el grafo.
        for k in range(len(Graph)):
            if v in Graph[k]:
                adjacencyNodes = Graph[k][v]
                break
            else:
                adjacencyNodes = None
        #Caso General de la recursividad.
        if adjacencyNodes != None:
            count = 0
            for j in adjacencyNodes:
                if j in listVisited:
                    count += 1
            if count is len(adjacencyNodes):
                Queue.pop(len(Queue)-1)
                count = 0
                #Ingersar los vértices donde se producen ciclos. (linea 131-136)
                for i in adjacencyList:
                    if v not in i:
                        count += 1
                if count == len(adjacencyList):
                    dictionary[v] = adjacencyListInDictionary
                    adjacencyList.append(dictionary)
                if len(Queue) != 0:
                    comeBackQueue = convertToDFSTreeRecursive(Graph,Queue[len(Queue)-1],listVisited,Queue,adjacencyList)
```

```python
            comeBackQueue = convertToDFSTreeRecursive(Graph,Queue[len(Queue)-1],listVisited,Queue,adjacencyList)
        else:
            comeBackQueue = convertToDFSTreeRecursive(Graph,Queue,listVisited,Queue,adjacencyList)
        return comeBackQueue
else:
    return "Vértice no conexo"
#Verificar que el vértice u sea conexo.
if adjacencyNodes is not None:
    #Recorrer la lista de adyacencia del vértice anterior(u).
    for i in adjacencyNodes:
    #Verificar que el vértice adyacente de u no se encuentre en la lista de visitados.
        if i not in listVisited:
            Queue.append(i)
            listVisited.append(i)
            adjacencyListInDictionary.append(i)
            dictionary[v] = adjacencyListInDictionary
            v = i
            break
#No agregar diccionarios vacíos a la lista del DFS.
if dictionary != {}:
    adjacencyList.append(dictionary)
```

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.

```python
def bestRoad(Graph,v1,v2):
    #Verificar que el grafo no esté vacío.
    if Graph == []:
        return "El grafo está vacío"
    #Verificar que los vértices v1 y v2 existen dentro del grafo.
    lista = []
    for i in range(len(Graph)):
        lista.append(list(Graph[i].keys()))
    if [v1] not in lista and [v2] not in lista:
        return f"El vértice {v1} y el vértice {v2} no se encuentran en el grafo."
    elif [v1] not in lista:
        return f"El vértice {v1} no se encuentra en el grafo."
    elif [v2] not in lista:
        return f"El vértice {v2} no se encuentra en el grafo."
    #Definir los nodos a utilizar.
    ListVertex = []
    for i in lista:
        listNodeVertex = insertVertex(i[0],ListVertex)
    #Definir el color de cada nodo.
    for node in listNodeVertex:
        node.color = "white"
        if node.value == v1:
            node.color == "gray"
    #Definir una cola.
    Queue = []
    Queue.append(v1)
    #Definir la lista de salida.
    listMinPath = []
    v2NodeVertex = nodeVertex()
    while Queue != []:
        vertex = Queue.pop(0)
        for node in listNodeVertex:
                if node.value == vertex:
                    nodeParent = node
                    node.color = "black"
        #Obtener los vértices adyacentes de vertex.
        adjacencyList = []
        for dicc in Graph:
            if vertex in dicc:
```

```python
    #Obtener los vértices adyacentes de vertex.
    adjacencyList = []
    for dicc in Graph:
        if vertex in dicc:
            adjacencyList.append(dicc[vertex])
    #Verificar que el vértice tenga vértices adyacentes o sea conexo.
    if adjacencyList != []:
        for vertexADJ in listNodeVertex:
            if vertexADJ.value in adjacencyList[0]:
                if vertexADJ.color == "white":
                    vertexADJ.color = "gray"
                    vertexADJ.parent = nodeParent
                    vertexADJ.distance = vertexADJ.parent.distance + 1
                    Queue.append(vertexADJ.value)
                    if vertexADJ.value == v2:
                        v2NodeVertex = vertexADJ
        #Agregar a la lista de salida el camino de v1 a v2.
        if v2NodeVertex.value == v2:
            while v2NodeVertex.distance != 0:
                listMinPath.insert(0,v2NodeVertex)
                v2NodeVertex = v2NodeVertex.parent
            listMinPath.insert(0,v2NodeVertex)
            return listMinPath
return []
```

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.

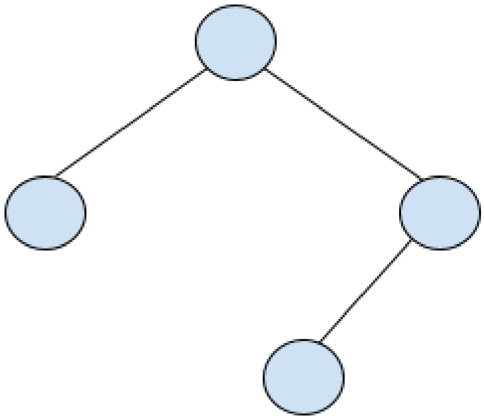
Ejercicio 12

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.

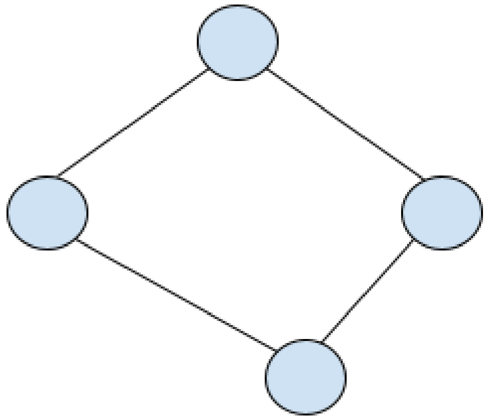
Sabiendo que por definición un árbol tiene k vértices y k-1 aristas, si se le agrega una arista nueva pasaría a tener k aristas por ende se formaría un ciclo en el árbol ya que esta nueva arista estaría comprendida por dos nodos del árbol y pasaremos de tener más de un camino para esos dos nodos.

Ejemplo:

Árbol:



Árbol con ciclo:



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.

Esto sucede ya que el BFS está planteado de esa manera si estamos en el nivel k con un vértice entonces en el nivel k+1 estarán los vértices adyacentes del vértice del nivel k. Es por eso que sus niveles nunca podrían diferenciarse en más de 1.

Parte 3

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
```

```python
def PRIM(Graph):
    #Verificar que el grafo no esté vacío.
    if Graph == []:
        return "El grafo está vacío."
    #Verificar que el grafo sea conexo.
    connectedGraph = isConnectedWeighted(Graph)
    if connectedGraph == False:
        return "El grafo no es conexo."
    #Obtener los vértices del grafo.
    listaVertex = []
    listaVertex.append(list(Graph[0].keys()))
    #Crear una lista para los vértices visitados.
    listVisited = []
    listVisited.append(listaVertex[0][0])
    #Crear lista de recurrencia para saber los caminos disponibles.
    listPath = []
    listPath.append(listaVertex[0][0])
    #Definir el árbol abarcador de costo mínimo.
    treeMinimum = []
    while len(listVisited) != len(Graph[0]):
        #Definir el diccionario que irá en la lista de adyacencia.
        dictionary = {}
        #Si el grafo tiene un solo vértice.
        if len(Graph[0]) == 1:
            dictionary[v] = None
            treeMinimum.append(dictionary)
            return treeMinimum
        #Recorrer una lista para encontrar el camino más corto.
        minWeightPrevius = 99999999999
        deleteVertex = None
        for v in listPath:
            count = 0
            #Obtener la lista de adyacencia del grafo.
            if v in Graph[0]:
                adjacencyNodes = Graph[0][v]
            #Obtener el menor peso.
            for weight in adjacencyNodes:
                if weight[0] not in listVisited:
                    if weight[1] <= minWeightPrevius:
                        minWeight = weight[1]
                        minWeightPrevius = minWeight
                else:
                    count += 1
            #Eliminar vértice de la lista ya que sus aristas no se tendran más en cuenta.
            if count == len(adjacencyNodes):
                deleteVertex = v
            else:
                #Recorrer la lista de adyacencia del vértice anterior(u).
                for vertex in adjacencyNodes:
                    #Verificar que el vértice adyacente de u no se encuentre en la lista de visitados.
                    if vertex[0] not in listVisited:
                        if vertex[1] <= minWeight:
                            u = vertex[0]
                            key = v
                            minWeight = vertex[1]
        listVisited.append(u)
        listPath.append(u)
        if deleteVertex != None:
            listPath.remove(deleteVertex)
        dictionary[key] = u
        treeMinimum.append(dictionary)
    return treeMinimum
```

Ejercicio 15

Implementar la función que responde a la siguiente especificación.

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

```python
def KRUSKAL(Graph):
    #Verificar que el grafo no esté vacío.
    if Graph == []:
        return "El grafo está vacío."
    #Verificar que el grafo sea conexo.
    connectedGraph = isConnectedWeighted(Graph)
    if connectedGraph == False:
        return "El grafo no es conexo."
    #Obtener los vértices del grafo.
    listVertex = []
    listVertex.append(list(Graph[0].keys()))
    #Crear las componentes conexas.
    listComponentConnected = []
    for vertex in listVertex[0]:
        LCC = make_set(vertex,listComponentConnected)
    #Obtener las aristas.
    listEdges = []
    for vertex in LCC:
        for edge in Graph[0][vertex]:
            edgeTypeList = list(edge)
            edgeTypeList.insert(0,vertex)
            listEdges.append(tuple(edgeTypeList))
    #Eliminar aristas repetidas
    listPositionDelete = []
    for tupleMain in range(len(listEdges)-1):
        for tupleSecond in range(tupleMain+1,len(listEdges)):
            if listEdges[tupleMain][0] == listEdges[tupleSecond][1] and listEdges[tupleMain][1] == listEdges[tupleSecond][0]:
                listPositionDelete.append(listEdges[tupleSecond])
    for tupleMain in listPositionDelete:
        listEdges.remove(tupleMain)
    #Ordenar las aristas de menor a mayor peso.
    SortEdgesWeight = sort_by_weight_asc(listEdges)

def make_set(vertex, LCC):
    LCC.append(vertex)
    return LCC
```

```python
def sort_by_weight_asc(listEdges): #BubbleSort
    count = 0
    while count < len(listEdges) - 1:
        for i in range(len(listEdges)-1):
            if listEdges[i][2] > listEdges[i+1][2]:
                deletedEdge = listEdges.pop(i)
                listEdges.insert(i+1,deletedEdge)
        count += 1
    return listEdges
```

No terminado

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.

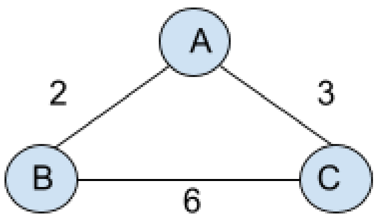
Esto es así ya que la arista <u,v> con u perteneciente a U y v perteneciente a V-U es una arista del AACM debido a que estos vértices pertenecen a Universos diferentes y por ende si estoy en U debería de buscar un vértice que no esté visitado anteriormente es decir que no pertenezca a U ya que si el vértice pertenece a U es porque ya se encontro el camino más corto que tiene como vértice a u.

Parte 4

Ejercicio 17

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

Ya sea la arista e cualquiera del grafo siguiente que forme un ciclo pero al ser la de mayor peso tomaremos <B,C>, está arista no pertenece al AACM debido a que no importa de qué vértice se parta se encontrará un camino de menor costo que incluya a B y C y no pasé por la arista <B,C> para todo AACM del grafo.

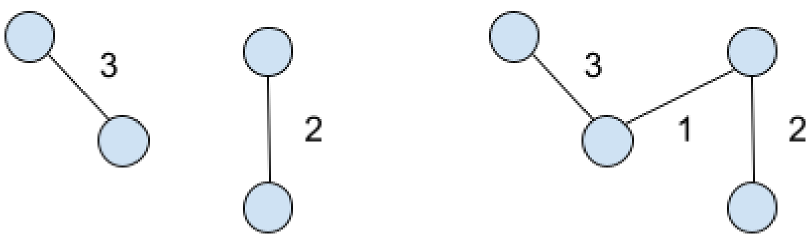


Ejercicio 18

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**)

Esto es así debido a que este es el funcionamiento del algoritmo de Kruskal por ejemplo:

Aquí tenemos dos AACM y si los unimos con una nueva arista de peso 1 ahora obtendremos un grafo con una componente conexa y un único AACM.



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.

Cambiar la manera de selección de aristas, en vez de tomar las de menor peso para obtener un AACMínimo hay que tomar las de mayor peso así obtener un AACMáximo.

2. Obtener un árbol de recubrimiento cualquiera.

No hacer diferencia sobre el peso de las aristas y usar el recorrido DFS.

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.

Dar el menor peso a las aristas de E y un peso mayor a las demás, así nuestro AACM estará formado por las aristas de E ya que sabemos que estás no forman ciclos.

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(V²)** 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.

Ya que todas las aristas tienen el mismo peso es indiferente el camino que se tome así que realizaría una búsqueda con BFS/DFS y por cada arista del árbol la agregaría a la matriz de VxV donde M[u, v] = 1 .

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.
```

```python
def shortestPath(Graph, s, v):  #Dijkstra
    if Graph == []:
        return "El grafo está vacío"
    #Verificar que los vértices v1 y v2 existen dentro del grafo.
    lista = []
    lista.append(list(Graph[0].keys()))
    if s not in lista[0] and v not in lista[0]:
        return f"El vértice {s} y el vértice {v} no se encuentran en el grafo."
    elif s not in lista[0]:
        return f"El vértice {s} no se encuentra en el grafo."
    elif v not in lista[0]:
        return f"El vértice {v} no se encuentra en el grafo."
    #Crear la lista de nodos.
    listNodes = []
    for i in range(len(lista[0])):
        listResultNodes = insertVertexDijkstra(lista[0][i], listNodes, i)
    #Init relax
    for vertexNode in listResultNodes:
        if vertexNode.value == s:
            initRelax(listResultNodes, vertexNode)
            break
    #Definir lista de nodos visitados.
    listVisited = []
    #Definir la cola.
    Queue = minQueue(listResultNodes)
    while Queue != []:
        #Obtener el vértice de la cola.
        u = Queue.pop(0)
        #Agregar el vértice a la lista de visitados.
        listVisited.append(u.value)
        #Obtener el camino de s a v.
        if u.value == v:
            #Si la distancia es la máxima es porque no existe un camino de s a v.
            if u.distance != 999999999:
                #Obtener el camino más corto de s a v.
                listShortestPath = []
                while u.parent != None:
                    listShortestPath.insert(0, u)
                    u = u.parent
```

```python
                    u = u.parent
                listShortestPath.insert(0, u)
                return listShortestPath
            else:
                return
        #Obtener los vértices adyacentes de u.
        adjacencyNodes = Graph[0][u.value]
        #Recorrer la lista de adyacencia de u.
        if adjacencyNodes != None:
            for vertex in adjacencyNodes:
                if vertex[0] not in listVisited:
                    relax(u,vertex, listResultNodes)
        else:
            return
        #Ordenar cola según la distancia en order ascendente
        minQueue(listResultNodes)
    return

def insertVertexDijkstra(v,ListNode,index):
    #Definir cada vértice del grafo como un nodo con sus atributos.
    vertexNode = nodeVertex()
    vertexNode.value = v
    vertexNode.key = index
    ListNode.append(vertexNode)
    return ListNode

def relax(u, tupleV, ListNodes):
    #Obtener el vértice adyacente de u en forma de nodo
    for node in ListNodes:
        if node.value == tupleV[0]:
            v = node
            break
    #Realizar relajo
    if v.distance > (u.distance + tupleV[1]):
        v.distance = u.distance + tupleV[1]
        v.parent = u
    return
```

```python
def minQueue(listNodes):
    count = 0
    #Ordenar la lista de nodos en orden ascendente según su distancia (usando BubbleSort).
    while count < len(listNodes) - 1:
        for i in range(len(listNodes)-1):
            if listNodes[i].distance > listNodes[i+1].distance:
                deletedNode = listNodes.pop(i)
                listNodes.insert(i+1,deletedNode)
        count += 1
    return listNodes
```

Ejercicio 22 (Opcional)

Sea **$G = \langle V, A \rangle$** un grafo dirigido y ponderado con la función de costos $C: A \rightarrow R$ de forma tal que $C(v, w) > 0$ para todo arco $\langle v, w \rangle \in A$. Se define el costo $C(p)$ de todo camino $p = \langle v_0, v_1, \ldots, v_k \rangle$ como $C(v_0, v_1) * C(v_1, v_2) * \ldots * C(v_{k-1}, v_k)$.

a) Demuestre que si $p = \langle v_0, v_1, \ldots, v_k \rangle$ es el camino de menor costo con respecto a C en ir de v_0 hacia v_k, entonces $\langle v_i, v_{i+1}, .., v_j \rangle$ es el camino de menor costo (también con respecto a C) en ir de v_i a v_j para todo $0 \leq i < j \leq k$.

b) ¿Bajo qué condición o condiciones se puede afirmar que con respecto a C existe camino de costo mínimo entre dos vértices $a, b \in V$? 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 $\langle v, w \rangle \in A$, proponga una función de costos $C': A \rightarrow R$ y además la forma de calcular el costo $C'(p)$ de todo camino $p = \langle v_0, v_1, \ldots, v_k \rangle$ 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.**~~